OSP-10546 US

日 本 国 特 許 庁
PATENT OFFICE
JAPANESE GOVERNMENT

US

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。
This is to certify that the annexed is a true copy of the following application as filed with this Office.

出願年月日
Date of Application: ２０００年 ４月２０日

出願番号
Application Number: 特願２０００－１１９５９４

願人
Applicant (s): 日本電気株式会社

CERTIFIED COPY OF
PRIORITY DOCUMENT

２００１年 ２月１６日

特許庁長官
Commissioner,
Patent Office 及川耕造

【書類名】　特許願
【整理番号】　37300360
【特記事項】　特許法第３６条の２第１項の規定による特許出願
【提出日】　平成12年 4月20日
【あて先】　特許庁長官　殿
【国際特許分類】　G06F 9/44
G06F 9/45
【発明者】
【住所又は居所】　東京都港区芝五丁目７番１号　日本電気株式会社内
【氏名】　マイルス　ゴードン　ベーダー
【特許出願人】
【識別番号】　000004237
【氏名又は名称】　日本電気株式会社
【代理人】
【識別番号】　100088328
【弁理士】
【氏名又は名称】　金田　暢之
【電話番号】　03-3585-1882
【選任した代理人】
【識別番号】　100106297
【弁理士】
【氏名又は名称】　伊藤　克博
【選任した代理人】
【識別番号】　100106138
【弁理士】
【氏名又は名称】　石橋　政幸
【手数料の表示】
【予納台帳番号】　089681
【納付金額】　35,000円

【提出物件の目録】

【物件名】　外国語明細書　1

【物件名】　外国語図面　1

【物件名】　外国語要約書　1

【包括委任状番号】　9710078

【プルーフの要否】　要

【書類名】　外国語明細書

1. Title of Invention

A method for avoiding excessive overhead while using a form of SSA (Static Single Assignment) extended to use storage locations other than local variables.

2. Claims

(1) A method for avoiding excessive overhead by a programmed computer while using a form of SSA (Static Single Assignment) extended to use storage locations other than local variables, comprising the step of:

allowing a program to use a compiler representation known as SSA form on any memory location addressable by the program; SSA form is normally only usable on function local variables.

(2) The method as claimed in claim 1, further comprising the steps of:

inserting phi functions at any place in the function where multiple definitions of a same non-SSA variable may be merged, the phi-functions producing a new definition of the variable at a point where they are inserted;

finding which operations may implicitly read or write complex variables that are in SSA form;

adding write-back copy operations at appropriate locations to write complex variables that are in SSA form, the write-back copy operations writing an SSA variable back to its real location;

adding read-back copy operations at appropriate locations to read possibly modified values back into new SSA definitions, the read-back copy operations defining a new SSA variable from a variable's real location; and

replacing every non-SSA variable definition by a definition of a unique SSA-variable, and replacing every non-SSA variable reference by a reference to an appropriate SSA-variable.

3. Detailed Description of Invention

Technical Field to which the Invention pertains

The present invention relates to Compiler Optimization.

Prior Art

This technique is related to an extension of the usual formulation of Static Single Assignment (SSA) form.

Briefly, `SSA form' is an alternative representation for variables in a program, in which any given variable is only assigned at a single location in the program. A program is transformed into SSA form by a process called `SSA conversion'. SSA conversion replaces every local variable in the source program with a set of new variables, called `SSA variables', each of which is only assigned to at a single physical location in the program -- thus, every point at which a source variable V is assigned to in the source program, the corresponding SSA-converted program will instead assign a unique variable, V'1, V'2, etc. At any point in the program (always at the start of a basic-block) where the merging of control flow would cause two such derived variables to be live simultaneously, their values are merged together to yielding a single new SSA variable, e.g., V'3, that represents the value of the
original source variable at that point. This merging is done using a `phi-function'. A phi-function is an instruction which has as many inputs as there are basic-blocks that can transfer control to the basic-block it is in, and chooses whichever input corresponds to the basic-block that preceded the current one in the dynamic control flow of the program.

SSA form is convenient because it allows variables to be treated as values, independent of their location in the program, making many transformations more straight-forward, as they don't need to worry about the implicit constraints imposed by using single variable

names to represent multiple values, depending on the location in the program. These properties make it a very useful representation for an optimizing compiler -- many optimization techniques become significantly simpler if the program is in SSA form. For instance, in a traditional compiler, a simple common-sub-expression-elimination algorithm that operates on variables must carefully guard against the possibility of redefinition of variables, so that it generally is only practical to use within a single basic block. However, if the program is in SSA form, this simple optimization need not worry about redefinition at all -- variables can't be redefined -- and furthermore will work even across basic block boundaries.

SSA conversion, as described above, is a transformation that is traditionally applied only to a function's local variables; this makes the process much easier, as local variables are subject to various constraints. For instance one knows that local variables are not aliased to other local variables, and unless its address has been taken, that a local variable will not be modified by a function other than the one it is declared in.

However, there are many cases where `active values', which one would like to receive the benefits of optimizations made possible by using SSA-form, exist in storage locations other than local variables. In this case, one would like to have the object's fields receive the same treatment as if they were a local variable, which could yield optimizations.

Information about SSA form can be found in the paper:

[SSAFORM] `Efficiently Computing Static Single Assignment Form and the Control Dependence Graph', by Ron Cytron et al., ACM TOPLAS, Vol. 13, No. 4, October 1991, Pages 451-490.

The SSA-conversion process in [SSAFORM] is performed in two steps [figure 12]:

(a) [201] Phi functions are inserted at any place in the function where multiple

definitions of the same non-SSA variable may be merged. The phi-functions produce a new definition of the variable at the point where they are inserted.

Because of this step, there is only one extant definition of a source variable at any point in the program.

(b) [202] Every non-SSA variable definition is replaced by a definition of a unique SSA-variable, and every non-SSA variable reference replaced by a reference to an appropriate SSA-variable -- because of the insertion of phi-functions, there will always be a single extant SSA-variable corresponding to a given non-SSA variable.

An extension of SSA form to non-local locations is described in:

[SSAMEM] `Effective Representation of Aliases and Indirect Memory Operations in SSA Form', by Fred Chow et al., Lecture Notes in Computer Science, Vol. 1060, April 1996, Pages 253-267.

The concept of basic-block `dominance' is well known, and can be described as follows:

A basic block A `dominates' a basic block B, if the flow of control can reach B only after A (although perhaps not immediately; other basics blocks may be executed between them).

If A dominates B and no other block dominates B that doesn't also dominate A, then A is said to be B's `immediate dominator'.

Problem to be solved by the Invention

It is desirable to extend the use of SSA form to handle non-local memory locations. However, a straight-forward implementation given the prior art, which synchronizes SSA

representations at every point of unknown behavior, can be very inefficient, because there are many operations that may read or write almost *any* memory location (for instance, in the case of library function calls, where the compiler often has no information about their behavior). Using such a simple technique also causes many extra phi-functions to be introduced, which can dramatically increase the cost of using SSA form.

This invention attempts to use SSA form on non-local memory locations, without excessive overhead for common program structures, by consolidating memory synchronization operations where possible.

Means for solving Problem

In this invention, we modify the procedure of [SSAFORM][figure 12] as follows:

Method for representing pointer variables in SSA form[452]:

+References or definitions of memory locations resulting from pointer-dereferences are also treated as `variables', here called `complex variables' [452], in addition to simple variables [451], such as those used in the source program. Complex variables consist of a pointer variable and an offset from the pointer. An example of a complex variable is the C source expression (lvalue) `*p', as used in [810] and [820].

Method for adding appropriate copy operations to synchronize complex variables [452] with the memory location they represent [figure 1]:

+These `complex variables' [452] are treated as non-SSA variables during SSA-conversion [figure 1] (any variable reference within a complex variable is treated as a reference in the instruction [440] that contains the complex variable [452]).

+A new step [120] is inserted in the SSA-conversion process [figure 1] between steps (a) [110] and (b) [130], to take care of any necessary synchronization of SSA-converted complex variables [452] with any instructions [440] that have unknown side-effects:

(a') [121] To any instruction [440] that may have unknown side-effects on an `active' complex variable [452] -- one that is defined by some dominator of the instruction -- add a list of the variable, and the possible side effects (may_read, may_write).

[122, 123] Next, insert special copy operations, called write-backs [521] (which write an SSA variable back to its real location) and read-backs (which define a new SSA variable from a variable's real location), to make sure the SSA-converted versions of affected variables [450] correctly synchronized with respect to such side-effects. This step may also insert new phi-functions, in the case where copying back a complex variable [452] from it's synchronization location may define a new SSA version of that variable.

For an example of adding write-backs [521] and read-backs [522], see [figure 4].

Mode for carrying out the Invention

This invention is an addition to a compiler for a computer programming language, whose basic control flow is illustrated in [figure 2]:

A source program [301] is converted into an internal representation by a parser [310], and if optimization is enabled, the internal representation is optimized by the optimizer [320]. Finally, the internal form is converted into the final object code [302] by the backend [330]. In a compiler that uses SSA form, the optimizer usually contains at least three steps: conversion of the program from the `pre-SSA' internal representation into an internal representation that uses SSA form [figure 1], optimization of the program in SSA form [322], and conversion of the program from SSA form to an internal representation without SSA form [323]. Usually SSA form differs from the non-SSA internal

representation only in the presence of additional operations, and certain constraints on the representation; see [SSAFORM] for details.

The preferred internal representation of a program used is as follows [figure 3]:

A program [410] is a set of functions.

A function [420] is a set of `blocks' [430], roughly corresponding to the common compiler concept of a `basic block'. A flow graph is a graph where the vertices are blocks [430], and the edges are possible transfers of control-flow between blocks [430]. A single block [430] is distinguished as the `entry block' [421], which is the block in the function executed first when the function is called.

Within a block [430] is a sequence of `instructions' [440], each of which describes a simple operation. Within a block [430], control flow moves between instructions [440] in the same order as their sequence in the block; conditional changes in control flow may only happen by choosing which edge to follow when choosing the successor block [432] to a block, so if the first instruction [440] in a block is executed, the others are as well, in the same
sequence that they occur in the block [430].

An instruction [440] may be a function call, in which case it can have arbitrary side-effects, but control-flow must eventually return to the instruction [440] following the function call.

An instruction [440] may explicitly read or write `variables' [450], each of which is either a `simple variable' [451], such as a local or global variable in the source program (or a temporary variable created by the compiler), or a `complex variable' [452], which represents a memory location that is indirectly referenced through another variable.

Each variable has a type, which defines what values may be stored in the variable.

Complex variables [452] are of the form `*(BASE + OFFSET)', where BASE [453] is a variable [450], and OFFSET [454] is a constant offset; this notation represents the value stored at memory location (BASE + OFFSET).

Because of the use of complex variables [452], there are typically no instructions [440] that serve to store or retrieve values from a computed memory location. Instead, a simple copy where either the source or destination, or both, is a complex variable [452] is used. Similarly, any other instruction [440] may store or retrieve its results and operands from memory using complex variables.

To assist in program optimization, each function is converted to SSA-form, which is described in (Prior Art) section, as modified for this invention, described in (Means for solving Problem). section. This conversion is called SSA-conversion, and takes place in 3 steps [figure 1], (a), (a'), and (b):

(a) [110] Phi functions are inserted at any place in the function where multiple definitions of the same variable may be merged, as described in [SSAFORM]. The phi-functions produce a new definition of the variable at the point where they are inserted. For example, the phi function [910] is inserted to merge the different values written to the complex variable `*p' at [911] ([820] in the input program) and [912] ([830] in the input program), and also at [1010], merging the values defined at [1011] ([820] in the input program) and [830] in the input program.

Because of this step, there is only one extant definition of a source variable at any point in the program.

(a') I. [121] For each operation, determine which `active' complex variables [452] it may

have unknown side-effects on, and list attach a note to the operation with this information. These notes are referred to below as `variable syncs'. In the example program, instructions [1020], [1021], [1022], and [1023] may possibly read or modify `*p' (as we don't have any information about them).

II. [122] At the same time, add any necessary write-back copy operations [521] write back any complex variables [452] to their `synchronization location'-- which is the original non-SSA variable (which, for complex variables [452], is a memory location), and mark the destination of the copy operation as such (this prevents step (b) of SSA conversion from treating the destination of the copy as a new SSA definition). Any such `write-back' [521] makes the associated variable inactive, and so prevents any further write-backs [521] unless the variable is once again defined.

III. [123] Add necessary read-backs, to supply new SSA definitions of complex variables [452] that have been invalidated (after having been written back to their synchronization location).

This is done by essentially solving a data-flow problem, where the values are `active read-backs', which are:

+ Defined by operations that may modify a complex variable [452], as located in step I above, or by the merging of multiple active read-backs of the same variable [450], at control-flow merge points. In the example, all the function calls may possibly modify `*p', so they must be represented by read-backs at [1020], [1021], [1022], and [1024].

+ Referenced by operations that use the value of a complex variable with an active read-back, or reaching a control-flow merge point at which no other read-backs of that variable are active (because such escaped definitions must then be merged with any other values of the complex variable using a phi-function).

Only read-backs that are referenced must actually be instantiated. In the example program, the only instantiated read-back is at [1030]. The reference that causes instantiation is the assignment of `*p' to the variable `x', at [840] in the source program; in the SSA-converted program, this assignment is split between the read-back at [1030] and the phi function at [1031].

+ Killed by definitions of the associated complex variable [452], or by a new read-back of the variable. In the example, the read-back defined at [1021] is killed because the following function call defines a new read-back of the same variable at [1022].

+ Merged, at control-flow merge points, with other active read-backs of the same variable [450], resulting in a new active read-back of the same variable. In the example, a `merge read-back' is defined at [1030], merging the read-backs of `*p' at [1022] and [1023].

After a fixed-point of read-back definitions is reached, those that are referenced are instantiated by inserting the appropriate copy operation at the place where they are defined, to copy the value from the read-back variable [450]'s synchronization location into a new SSA variable; if necessary new phi-functions may be inserted to reflect this new definition point. As
mentioned above, in the example this only happens at [1030].

Steps (a'.I) [121] and (a'.II) [122] take place as follows:

Call the procedure `add_syncs_and_write_backs' [figure 5] on the function's entry block [430], initializing the ACTIVE_VARIABLES and ALL_ACTIVE_VARIABLES parameters to empty lists.

The procedure `add_syncs_and_write_backs', with arguments BLOCK, ACTIVE_VARIABLES, and ALL_ACTIVE_VARIABLES is defined as follows [figure 5]:

[610] For every instruction [440] in the BLOCK, do:

[620] For each VARIABLE in ALL_ACTIVE_VARIABLES, do:

[621] If INSTRUCTION may possibly read or write VARIABLE, then [622] add a `variable sync' describing the possible reference or modification to INSTRUCTION.

[625] If INSTRUCTION may possibly read or write VARIABLE, and is also in ACTIVE_VARIABLES, then [626] add a `write-back' copy operation just before INSTRUCTION to write VARIABLE back to its synchronization location, and [627] remove VARIABLE from ACTIVE_VARIABLES. Because at this stage of SSA conversion, only source variables are present (not SSA variables), then this write-back copy operation is represented by a copy from VARIABLE to itself (`VARIABLE := VARIABLE') with a special flag set to indicate that the destination should not be SSA-converted.

[630] For each VARIABLE which is defined in INSTRUCTION, do:

Add VARIABLE to ACTIVE_VARIABLES and ALL_ACTIVE_VARIABLES (modifications to these variables are local to this function).

[650] For each block [430] immediately dominated by BLOCK, DOM, do:

[651] Recursively use add_syncs_and_write_backs on the dominated block DOM, with the local values of ACTIVE_VARIABLES and ALL_ACTIVE_VARIABLES passed as

the respectively named parameters.

Step (a'.III) take place as follows [figure 6]:

[701] Initialize the mappings BLOCK_BEGIN_READ_BACKS and BLOCK_END_READ_BACKS to be empty. These mappings associate each block in the flow graph with a sets of read-backs.

[702] Initialize the queue PENDING_BLOCKS to the function's entry block.

[710] While PENDING_BLOCKS is not empty, [711] remove the first block [430] from it, and invoke the function `propagate_block_read_backs' [800] on that block.

[720] For each read-back RB in any block [430] that has been marked as `used', and [721] isn't a `merge read-back' who's sources (the read-backs that it merges) are all also marked `used', instantiate that read-back as follows:

[730] If RB is a `merge read-back', then the point of read-back is [741] the beginning of the block [430] where the merge occurs, otherwise it is [742] immediately after the instruction [440] that created the read-back.

[731] Add a copy operation at the point of read-back that copies RB's variable from its synchronization location to an SSA variable (as noted above for adding write-back copy operations, because at this stage no SSA variable have actually been introduced, this copy operation simply copies from the variable to itself, but marks the source of the copy with a flag saying not to do SSA conversion).

[732] If necessary, introduce phi functions to merge the newly defined SSA variable with other definitions of the variable.

The function `propagate_block_read_backs', with the parameter BLOCK, is defined as follows [figure 7]:

[801] Look up BLOCK in BLOCK_BEGIN_READ_BACKS and BLOCK_END_READ_BACKS, assigning the associated read-back set with the local variables OLD_BEGIN_READ_BACKS and OLD_END_READ_BACKS respectively. If there is no entry for block in either case, add an appropriate empty entry for block.

[810] Calculate the intersection of the end read-back sets for each predecessor block [431] of BLOCK in the flow-graph, calling the result NEW_BEGIN_READ_BACKS. The intersection is calculated as follows:

Any predecessor read-back for which a read-back of the same variable doesn't exist in one of the other predecessor blocks is discarded from the result; it is also marked as `referenced'.

If the read-back for a given variable is the same read-back in all predecessor blocks [431], that read-back is added to the result.

If a given variable is represented by different read-backs in at least two predecessor blocks [431], a `merge read-back' is created that references all the corresponding predecessor read-backs, and this merge read-back is added to the result.

[820] If NEW_BEGIN_READ_BACKS is different from OLD_BEGIN_READ_BACKS, or this is the first time this block has been processed, then:

[821] Add NEW_BEGIN_READ_BACKS as the entry for BLOCK in

BLOCK_BEGIN_READ_BACKS, replacing OLD_BEGIN_READ_BACKS.

[822] Initialize NEW_END_READ_BACKS from NEW_BEGIN_READ_BACKS.

[830] For each operation INSTRUCTION in BLOCK, do:

[840] For each variable reference VREF in INSTRUCTION, do:

[845] If VREF has an entry RB in NEW_END_READ_BACKS, then [846] Mark RB as used, and [847] remove it from NEW_END_READ_BACKS.

[850] For each variable definition VDEF in INSTRUCTION, do:

[855] If VDEF has an entry RB in NEW_END_READ_BACKS, then [856] remove RB from NEW_END_READ_BACKS.

[860] For each variable sync in INSTRUCTION that notes a variable VARIABLE as possibly written, do:

[865] Add a new read-back entry for VARIABLE to NEW_END_READ_BACKS, replacing any existing read-back of VARIABLE.

[870] If NEW_END_READ_BACKS is different from OLD_END_READ_BACKS, then:

[871] Add NEW_END_READ_BACKS as the entry for BLOCK in BLOCK_END_READ_BACKS, replacing OLD_END_READ_BACKS.

[880] Add each BLOCK's successors [432] to PENDING_BLOCKS.

(b) [130] Every non-SSA variable definition is replaced by a definition of a unique SSA-variable, and every non-SSA variable reference replaced by a reference to an appropriate SSA-variable, as described in [SSAFORM].

The exception to this rule is complex variables [452] that have been marked as special `synchronization' locations, in the copy instruction [440] inserted in step (a'); they are left as-is, referring to the original complex variable [452].

An example of a program being transformed into SSA form, with and without the use of this invention, can be found in figures 8 - 11.

Effect of the Invention

This invention adds synchronization operations that allow the efficient use of SSA-form for non-local memory locations in the presence of possible aliasing.

4. Brief Description of Drawings

Figure 1 shows general form of the SSA-conversion process used by this invention.

Figure 2 shows overall compiler control flow.

Figure 3 shows basic data structures used in describing this invention.

Figure 4 shows placement of variable read- and write-backs.

Figure 5 shows the control flow of the procedure for steps (a'.I) and (a'.II) of the modified SSA conversion process, adding variable synchronization information to instructions and adding variable write-backs to a function, `add_syncs_and_write_backs'.

Figure 6 shows the control flow of the procedure for step (a'.III) of the modified SA conversion process, adding variable read-backs to a function.

Figure 7 shows the control flow for a subroutine used by step (a'.III) of the modified SSA conversion process, `add_merged_read_backs'.

Figure 8 shows example source program.

Figure 9 shows SSA converted program, with simple implementation of read-backs.

Figure 10 shows SSA converted program, with the implementation of read-backs described in this patent.

Figure 11 shows register-allocated and SSA-unconverted program.

Figure 12 shows general form of the traditional SSA-conversion process.

Description of the Reference Numerals

301 Input source file

320 Optimizer

302 Output file

410 Program

420 Function

421 Entry block

430 Block

440 Instruction

450 Variable

【書類名】　外国語図面

Figure 1. Modified SSA-conversion process

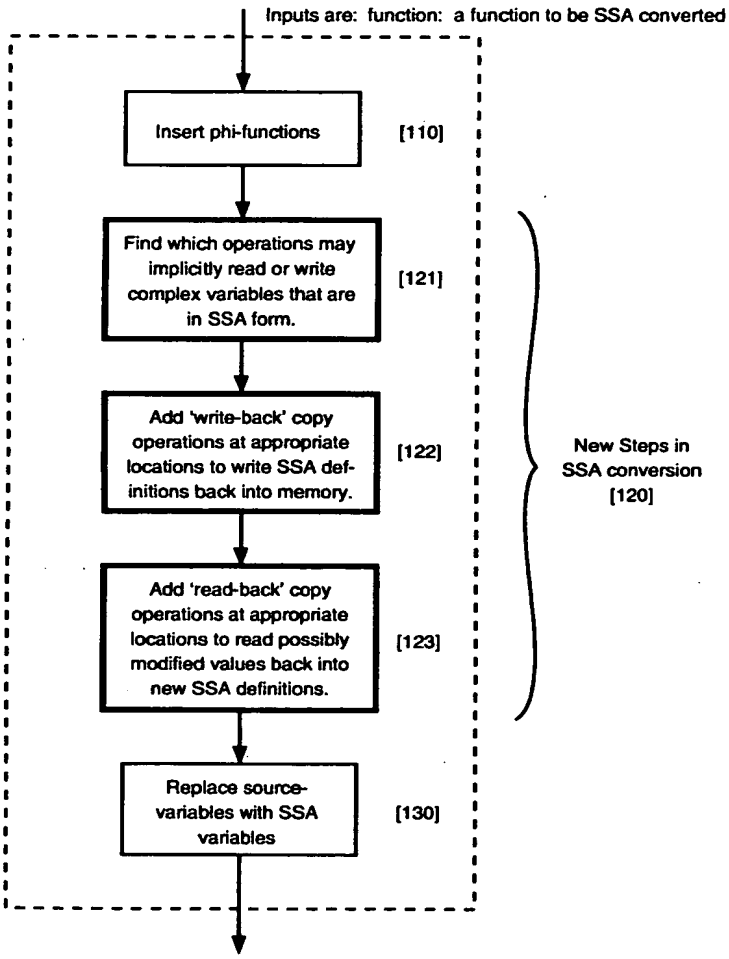


Figure 2. Overall compiler control flow

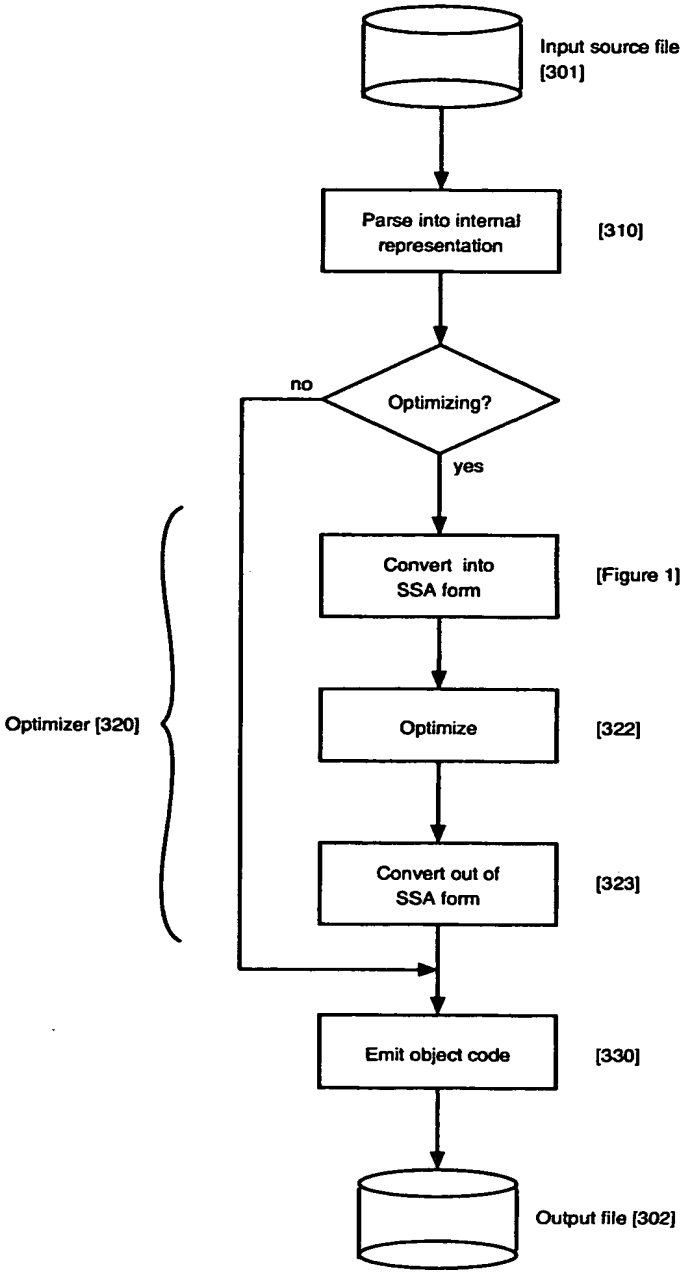


Figure 3. Program representation

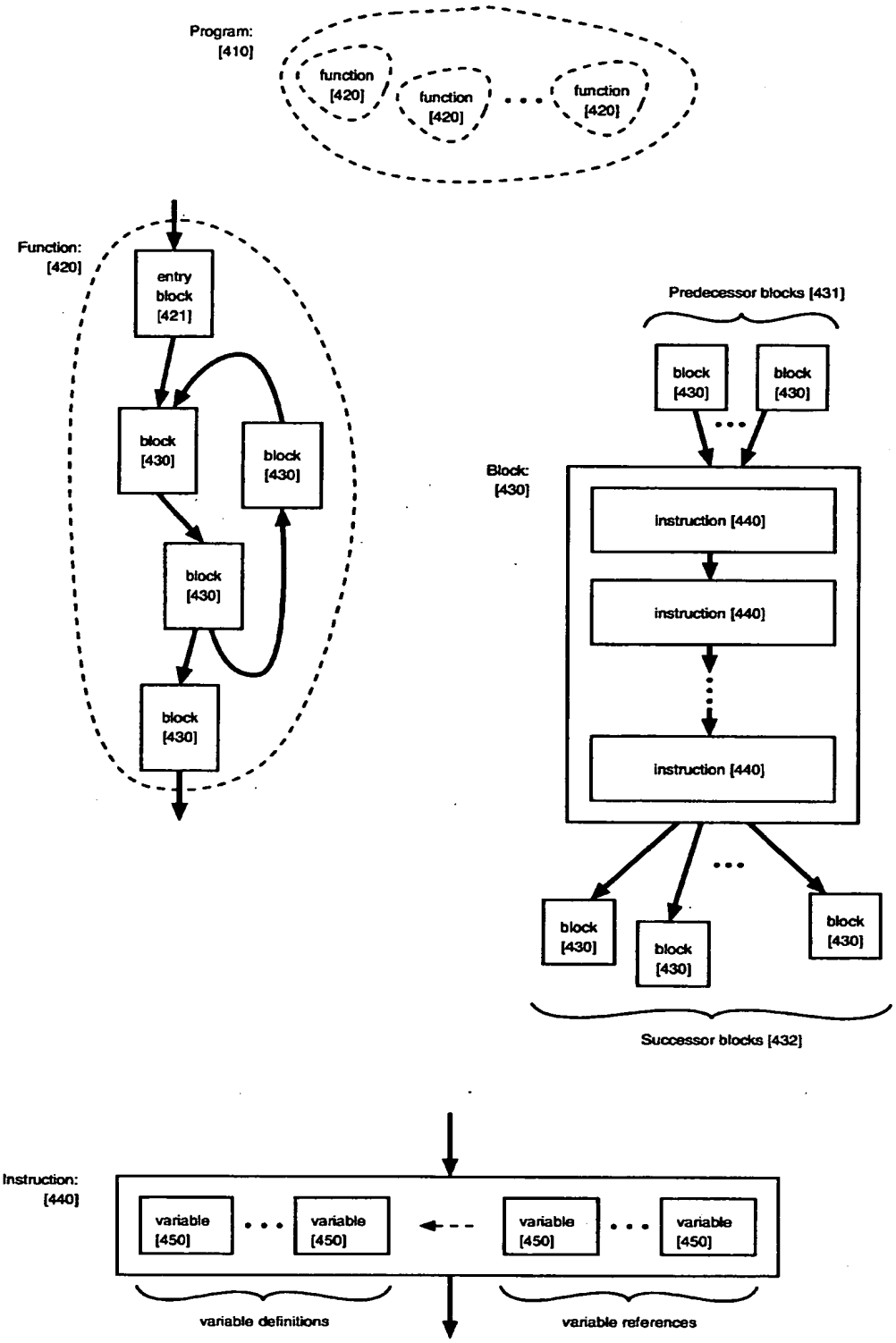


Figure 4. Placement of read/write-backs for the SSA form of *x, (*x)'1

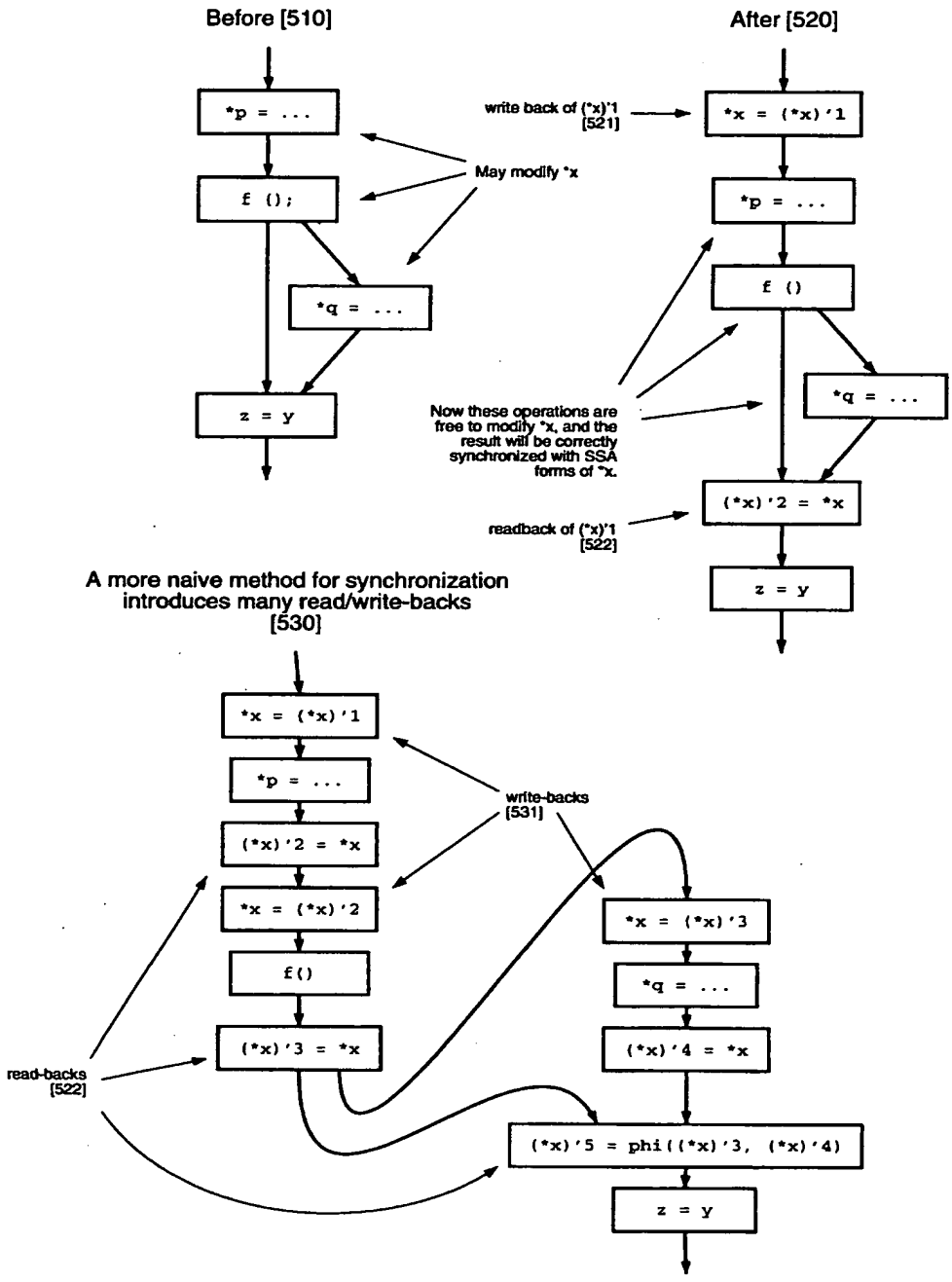


Figure 5. The procedure 'add_syncs_and_write_backs'

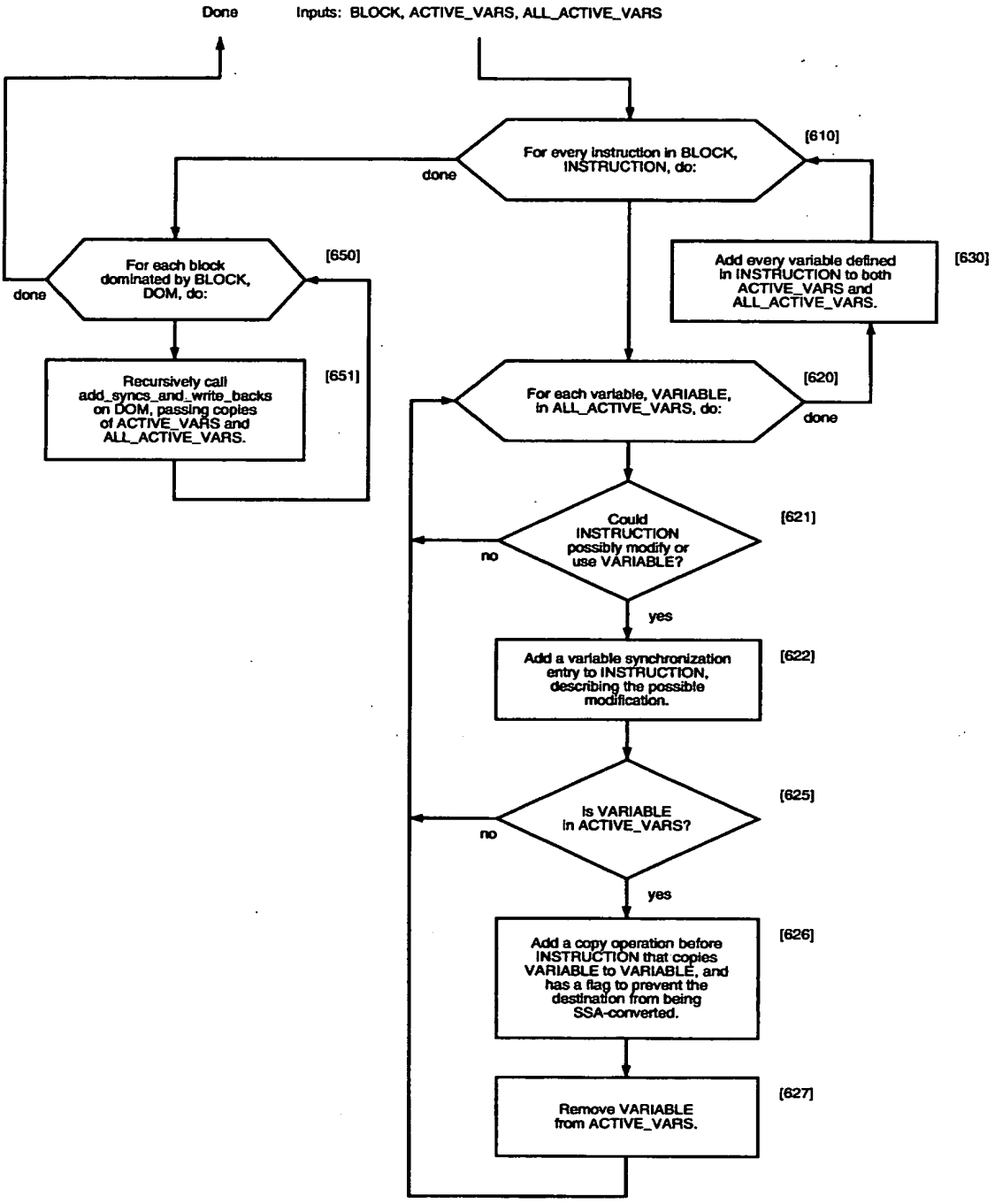


Figure 6. Conversion step (a'.III), insertion of read-backs

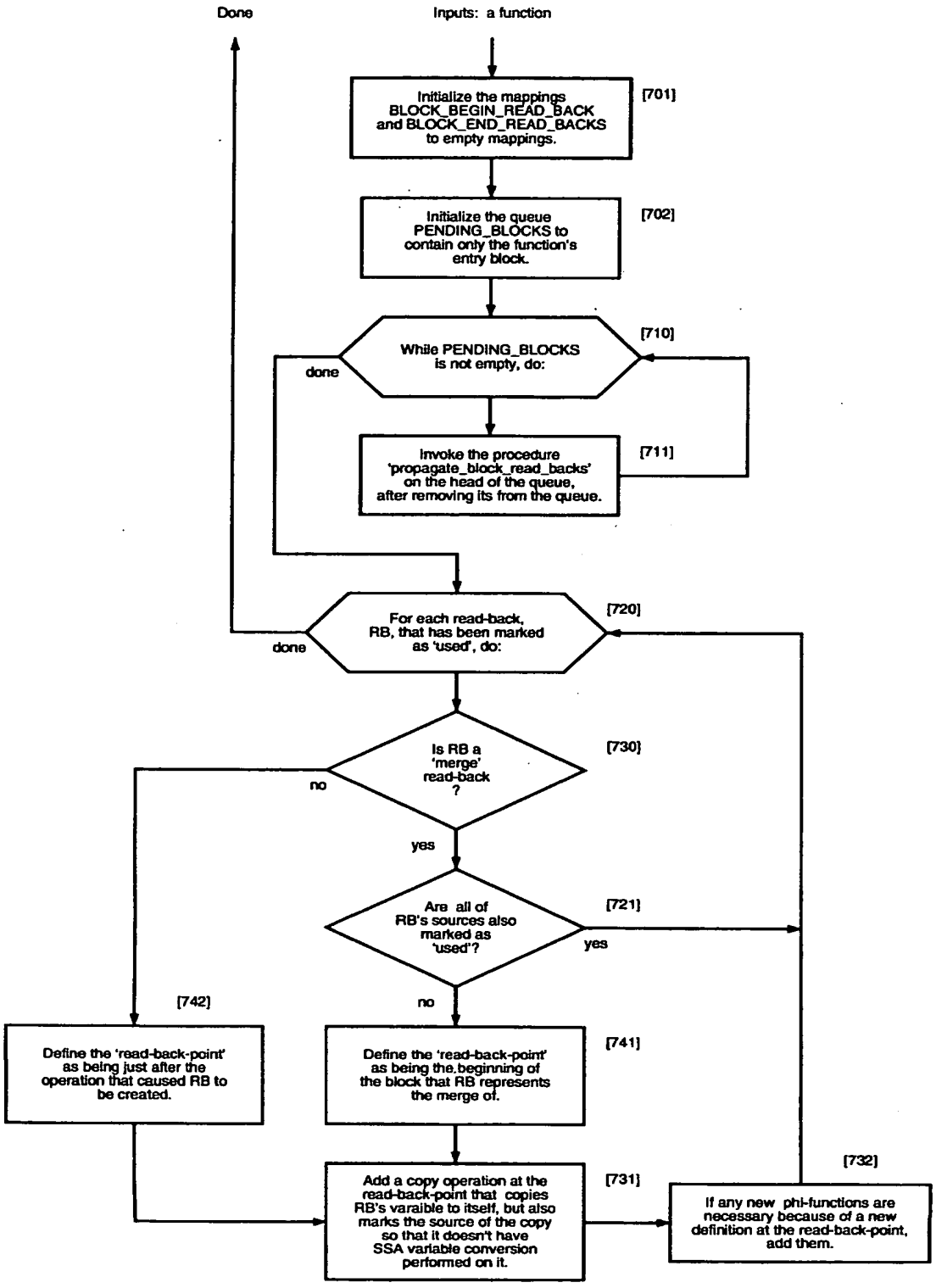


Figure 7. The procedure 'propagate_read_backs'

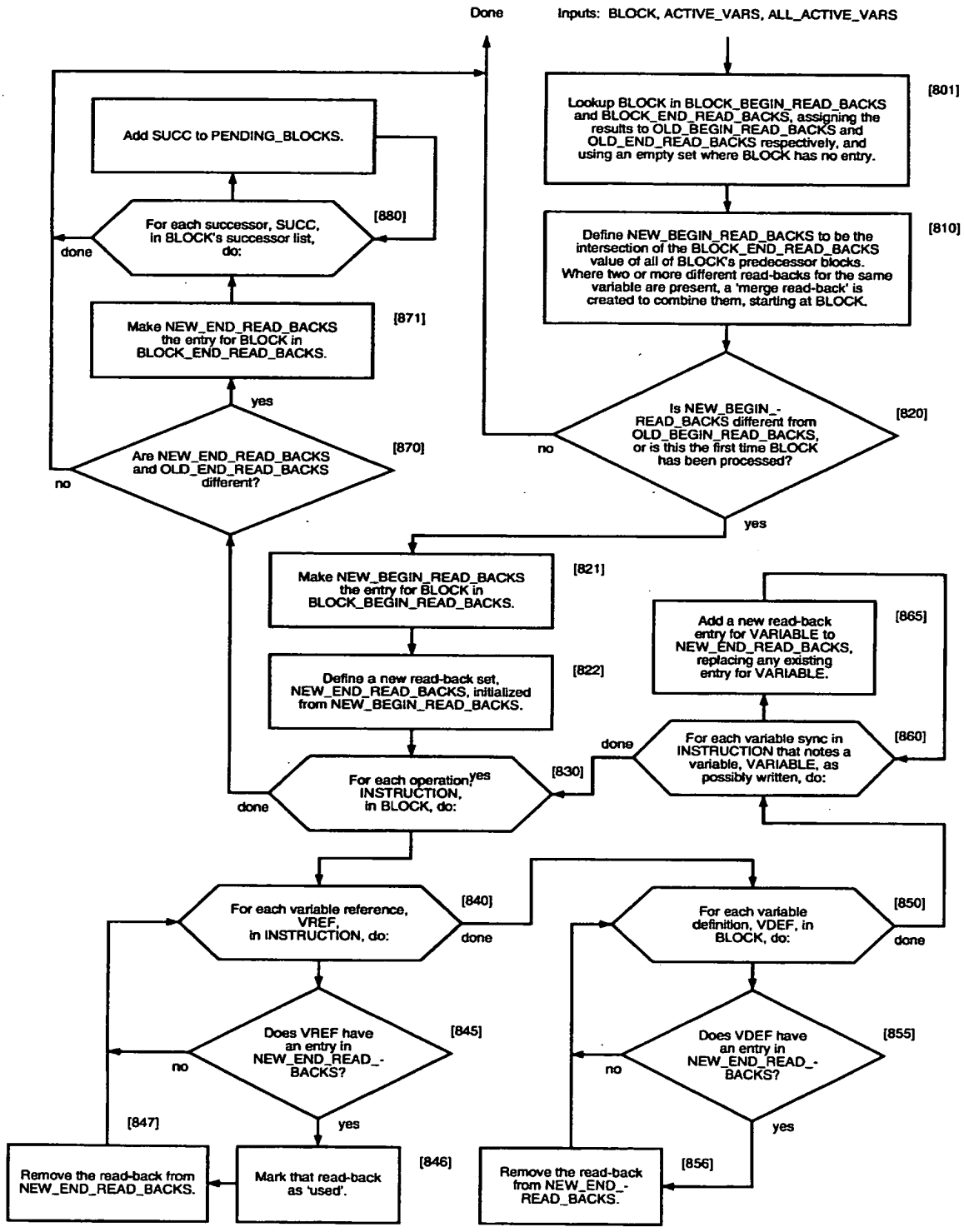


Figure 8: Example source program

This short C program is used to illustrate the invention:

```
extern int g (), h (), i (), x;
int foo (int *p)
{
  (*p)++;                                                    [810]
  if (*p > 10)
    {
      g ();
      h ();
      if (x > 5)
        g ();
      if (x > 3)
        i ();
      else
        x = *p;
      *p = 5;
    }
  return *p;
}
```

Here's the same program converted to a slightly more primitive form:

```
int foo (int *p)
{
block1:
  *p := *p + 1;                                              [820]
  if (*p <= 10)
    goto block8;

block2:
  g ();
  h ();
  if (x <= 5)
    goto block4;

block3:
  g ();

block4:
  if (x > 3)
    goto block6;

block5:
  x := *p;                                                   [840]
  goto block7;

block6:
  i ();

block7:
  *p := 5;                                                   [830]

block8:
  return *p;
}
```

Figure 9: SSA converted program, with simple implementation of read-backs:

The following is psuedo-C, augmented with the 'phi' operation, where

RESULT = phi (block1: VAL1, ..., blockN: VALN)

means 'assign VAL1 to RESULT if control-flow comes from block1', and similarly so on for each value of N.

The extra variables "pvN", where N is an integer, are SSA versions of *p, and are in fact local variables, not dereferences of p.

```
int foo (int *p)
{
  int pv1, pv2, pv3, pv4, pv5, pv6;

block1:
  pv1 = *p + 1;
  if (pv1 <= 10)
    goto block8;

block2:
  *p = pv1;          /* This writes-back PV1 to *P.   */
  g ();
  pv2 = *p;          /* This reads-back *P into PV2.  */
  *p = pv2;          /* This writes-back PV2 to *P.   */
  h ();
  pv3 = *p;          /* This reads-back *P into PV3.  */      [912]
  if (x <= 5)
    goto block4;

block3:
  *p = pv3;          /* This writes-back PV4 to *P.   */
  g ();
  pv4 = *p;          /* This reads-back *P into PV4.  */      [911]

block4:
  pv5 = phi (block3: pv4, block2: pv3)                        [910]
  if (x > 3)
    goto block6;

block5:
  goto block7;

block6:
  i ();

block7:
  x = phi (block6: x, block5: pv5);

block8:
  pv6 = phi (block1: pv1, block7: 5);
  *p = pv6;          /* This writes-back PV6 to *P.   */

  return pv6;
}
```

Figure 10: SSA converted program, with the implementation of read-backs described in this patent

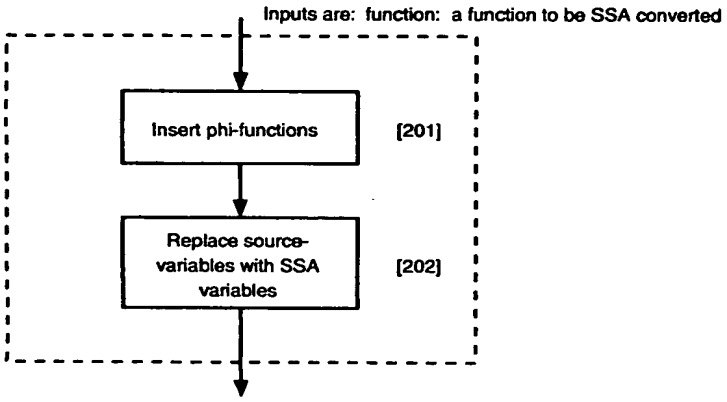
```
int foo (int *p)
{
  int pv1, pv2, pv3;

block1:
  pv1 = *p + 1;                                                    [1011]
  if (pv1 <= 10)
    goto block8;

block2:
  *p = pv1;          /* This writes-back PV1 to *P.  */
  g ();                                                            [1021]
  h ();                                                            [1022]
  if (x <= 5)
    goto block4;

block3:
  g ();                                                            [1023]

block4:
  pv2 = *p;          /* This reads-back *P into PV2.  */           [1030]
  if (x > 3)
    goto block6;

block5:
  goto block7;

block6:
  i ();                                                            [1024]

block7:
  x = phi (block6: x, block5: pv2);                                [1031]

block8:
  pv3 = phi (block1: pv1, block7: 5);                              [1010]
  *p = pv3;          /* This writes-back PV3 to *P.  */

  return pv3;
}
```

Figure 11: Register-alloced and SSA-unconverted program

Using SSA-form requires having a good register allocator that will merge variables where possible, as it tends to generate a lot of variables with short lifetimes. We assume that here.

```
int foo (int *p)
{
  int pv;

block1:
  pv = *p + 1;
  if (pv <= 10)
    goto block8;

block2:
  *p = pv;          /* This writes-back PV to *P.  */
  g ();
  h ();
  if (x <= 5)
    goto block4;

block3:
  g ();

block4:
  if (x > 3)
    goto block6;

block5:
  x = *p;
  goto block7;

block6:
  i ();

block7:
  pv = 5;

block8:
  *p = pv;          /* This writes-back PV to *P.  */

  return pv;
}
```

Figure 12. Original SSA-conversion process



【書類名】　外国語要約書

1. Abstract

subjective

The usual formulation of the compiler representation known as `SSA-form' can only handle local variables. It is desirable to extend this to allow other locations to be represented.

means for solution

This invention adds synchronization operations that allow the efficient use of SSA-form for non-local memory locations in the presence of possible aliasing.

2. Representative Drawing

Figure 1

【書類名】　翻訳文提出書

【整理番号】　37300360

【提出日】　平成12年 6月 9日

【あて先】　特許庁長官　　殿

【出願の表示】

　【出願番号】　特願2000-119594

【特許出願人】

　【識別番号】　000004237

　【氏名又は名称】　日本電気株式会社

【代理人】

　【識別番号】　100088328

　【弁理士】

　【氏名又は名称】　金田　暢之

　【電話番号】　03-3585-1882

【確認事項】　本書に添付した翻訳文は、外国語書面出願の願書に添付して提出した外国語明細書、外国語図面及び外国語要約書に記載した事項を過不足なく適正な日本語に翻訳したものである。

【提出物件の目録】

　【物件名】　外国語明細書の翻訳文　1

　【物件名】　外国語図面の翻訳文　1

　【物件名】　外国語要約書の翻訳文　1

【プルーフの要否】　要

【書類名】　明細書

【発明の名称】　局所変数以外の記憶位置を用いるように拡張されたＳＳＡ形式を使用して、過度のオーバーヘッドを避ける方法

【特許請求の範囲】

【請求項１】　ＳＳＡ（静的単一割り当て）形式は関数の局所変数のみに通常は使用可能であるのに対し、プログラムがＳＳＡ形式として知られるコンパイラ表現を、前記プログラムによりアドレス可能な任意の記憶場所に使用することを可能とする段階を有する、局所変数以外の記憶位置を用いるように拡張されたＳＳＡの形式を使用して、プログラムされたコンピュータにより過度のオーバーヘッドを避ける方法。

【請求項２】　ファイ関数は前記ファイ関数が挿入された点に同一の非ＳＳＡ変数の新しい定義を生成する関数であり、前記同一の非ＳＳＡ変数の複数の定義をマージすることができる前記ファイ関数の任意の位置に前記ファイ関数を挿入する段階と、

どの操作が、ＳＳＡ形式であるコンプレックス変数を、暗黙的に読み取り又は書き込みすることができるかを検出する段階と、

ライトバック・コピー操作はＳＳＡ変数をその実際の位置に書き戻す操作であり、ＳＳＡ形式であるコンプレックス変数を書き込むために適切な位置に前記ライトバック・コピー操作を追加する段階と、

リードバック・コピー操作は変数の実際の位置から新しいＳＳＡ変数を定義する操作であり、新しいＳＳＡ定義の中に修正された可能性のある値をリードバックするために適切な位置に前記リードバック・コピー操作を追加する段階と、

すべての非ＳＳＡ変数定義をユニークなＳＳＡ変数の定義により置換し、すべての非ＳＳＡ変数レファレンスを適切なＳＳＡ変数へのレファレンスにより置換する段階とをさらに有する請求項１記載の方法。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】

本発明はコンパイラの最適化に関する。

【０００２】

【従来の技術】

本技術は静的単一割り当て（ＳＳＡ）形式の通常の公式化の拡張に関する。

【０００３】

簡潔に言えば、’ＳＳＡ形式’はプログラムにおける変数に対する別の表現である。’ＳＳＡ形式’において、変数はそのプログラム内で単一の位置にのみ割り当てられる。プログラムは’ＳＳＡ変換’と呼ばれるプロセスによりＳＳＡ形式に変換される。ＳＳＡ変換はソース・プログラム内のすべての局所変数を’ＳＳＡ変数’と呼ばれる一組の新しい変数に置換する。’ＳＳＡ変数’のそれぞれは、そのプログラム内で単一の物理的な位置、したがって、原変数Ｖがソース・プログラム内で割り当てられるすべての点にのみ割り当てられる。対応するＳＳＡ変換されたプログラムは、ユニークな変数Ｖ’１、Ｖ’２などを代わりに割り当てるであろう。制御流れのマージングがこのような２つの導出された変数を同時に生変数とさせるプログラム内の任意の点（常に基本ブロックの開始点）において、２つの導出された変数の値は１つにマージされ、その時点で元の原変数の値を表す単一の新しいＳＳＡ変数、たとえば、Ｖ’３を生ずる。このマージングは’ファイ関数’を使用して行われる。ファイ関数は、命令がある基本ブロックに制御を転送することができる基本ブロックと同数の入力を有し、プログラムの動的な制御流れの中で現在の入力に先行する基本ブロックにどの入力が対応するかを選択する命令である。

【０００４】

ＳＳＡ形式は、プログラム内の位置によって複数の値を表すために単一の変数名を使用することにより課される暗黙の制約条件を心配する必要がないので、プログラム内の変数の位置から独立して変数が値として処理されることを可能にし、多くの変換をより直接的にするので、ＳＳＡ形式は都合が良い。これらの特性は最適化コンパイラにとってＳＳＡ形式を非常に有用な表現にする。プログラムがＳＳＡ形式ならば、多くの最適化技術は大幅に簡単になる。たとえば、従来のコンパイラでは、変数の上で操作する単純な共通式の削除アルゴリズムは、変数の再定義の可能性を注意深く保護しなければならない。したがって、従来のコン

パイラは単一の基本ブロックの中でのみ使用することが通常実際的である。しかし、プログラムがＳＳＡ形式であれば、この単純な最適化は再定義について心配する必要はまったくない。変数は再定義できず、さらに基本ブロック境界を越えて動作する。

【０００５】

上述のように、ＳＳＡ変換は関数の局所変数にのみ伝統的に応用されていた変換である。局所変数は様々な制約条件に支配されるから、ＳＳＡ変換は処理を非常に容易にする。たとえば、局所変数は他の局所変数に別名を付けないことは知られており、局所変数のアドレスがとられないかぎり、局所変数はそれが宣言された関数以外の関数により修正されないことは知られている。

【０００６】

しかし、ＳＳＡ形式を使用することにより可能となる最適化の利益を受けるために望ましい、’アクティブな値’が局所変数以外の記憶位置に存在する多くの場合がある。この場合に、オブジェクトのフィールドが局所変数であるかのように最適化を生ずる同じ取り扱いを受けることが好ましい。

【０００７】

ＳＳＡ形式についての情報は下記の論文に見いだすことができる。

【０００８】

［ＳＳＡＦＯＲＭ］、’静的単一割り当て形式及び制御依存性グラフの効率的な計算’、ロン・サイトロン他、ＡＣＭ　ＴＯＰＬＡＳ、Ｖｏｌ．１３、Ｎｏ．４、１９９１年１０月、４５１－４９０ページ。

【０００９】

［ＳＳＡＦＯＲＭ］内のＳＳＡ変換処理は２つの段階で行われる［図１２］。

【００１０】

（ａ）［２０１］、同一の非ＳＳＡ変数の複数の定義をマージすることができる関数内の任意の位置にファイ関数が挿入される。ファイ関数は、ファイ関数が挿入される点に変数の新しい定義を作る。

【００１１】

この段階のために、プログラム内の任意の点において、原変数の実在の定義は

１つのみ存在する。

【００１２】

（ｂ）［２０２］、ファイ関数の挿入のために、すべての非ＳＳＡ変数の定義は固有のＳＳＡ変数の定義により置換され、すべての非ＳＳＡ変数レファレンスは適切なＳＳＡ変数へのレファレンスにより置換される。一定の非ＳＳＡ変数に対応して単一の実在のＳＳＡ変数が常に存在する。

【００１３】

ＳＳＡ形式の非局所的な位置への拡張は下記の文献に説明されている。

【００１４】

［ＳＳＡＭＥＭ］、’ＳＳＡ形式における別名と間接的なメモリ操作の有効な表現’、フレッド・チョー他、コンピュータ・サイエンス講義ノート、Ｖｏｌ．１０６０、１９９６年４月、２５３－２６７ページ。

【００１５】

基本ブロック’優位’の概念は公知であり、次のように説明できる。

【００１６】

制御の流れがＡの後にＢに到達することができる場合にのみ、基本ブロックＡは基本ブロックＢの’優位に立つ’、（恐らく直ちにではないが、他の基本ブロックはそれらの間に実行されることができる）。

【００１７】

ＡがＢの優位に立ち、さらにＡの優位に立たない他のどのブロックもＢの優位に立たなければ、ＡはＢの’イミーディエイト・ドミネータ’である。

【００１８】

【発明が解決しようとする課題】

非局所的記憶場所を処理するために、ＳＳＡ形式の使用を拡張することが望ましい。しかし、未知の操作のすべての点においてＳＳＡ表現を同期させる従来技術による直接的な具体化例は、ほとんどすべての記憶場所を読み取りあるいは書き込みする可能性のある多くの操作があるために、（たとえば、多くの場合コンパイラがその操作についての情報を有しないライブラリ関数呼出しの場合）非常に非効率的な恐れがある。このような単純な技術を使用することは、多くの追加

のファイ関数を導入させる原因ともなり、ＳＳＡ形式を使用するコストを劇的に増加させる恐れがある。

【００１９】

本発明は、メモリ同期操作を可能な限り強化することにより、普通のプログラム構造に対して過度のオーバーヘッドなしに、非局所的記憶場所にＳＳＡ形式の使用を試みる。

【００２０】

【課題を解決するための手段】

本発明において、処理手順は次のように修正される。［ＳＳＡＦＯＲＭ］［図１２］

ＳＳＡ形式内のポインタ変数を表す方法。［４５２］

ソース・プログラムで使用されるような単純な変数に加えて［４５１］、ポインタ・デレファレンスから生ずる記憶場所の参照あるいは定義も、ここで’コンプレックス変数’と呼ばれる’変数’として処理される［４５２］。コンプレックス変数は、ポインタ変数及びポインタからのオフセットから成る。コンプレックス変数の例は、［８１０］及び［８２０］で使用されるような、Ｃソース式（レフトバリュー）の’＊ｐ’である。

【００２１】

コンプレックス変数が表す記憶場所に、コンプレックス変数［４５２］を同期させる適切なコピー操作を追加する方法。［図１］

これらの’コンプレックス変数’［４５２］は、ＳＳＡ変換の間に非ＳＳＡ変数として処理される［図１］（コンプレックス変数内のすべての変数レファレンスは、コンプレックス変数［４５２］を含む命令［４４０］の中のレファレンスとして処理される）。

【００２２】

未知の副作用を有するすべての命令［４４０］を有するＳＳＡ変換されたコンプレックス変数［４５２］の必要なすべての同期の処理をするために、新しい段階［１２０］が、段階（ａ）［１１０］と段階（ｂ）［１３０］の間に、ＳＳＡ変換処理［図１］の中に挿入される。

【００２３】

（ａ’）［１２１］、命令のいくつかのドミネータにより定められる’アクティブな’コンプレックス変数［４５２］に未知の副作用を有する恐れのあるすべての命令［４４０］に対して、変数のリスト及び可能性のある副作用（may_read、may_write）を追加する。

【００２４】

［１２２、１２３］次に、影響を受けた変数［４５０］のＳＳＡ変換されたバージョンが、このような副作用に対して正確に同期したことを確認するために、ライトバック［５２１］（ＳＳＡ変数をその実際の位置に書き戻す）、及びリードバック（変数の実際の位置から新しいＳＳＡ変数を定める）と呼ばれる特別なコピー操作を挿入する。この段階は、コンプレックス変数の同期位置からコンプレックス変数［４５２］をコピー・バックすることが、その変数の新しいＳＳＡバージョンを定める可能性がある場合には、新しいファイ関数をさらに挿入することもできる。

【００２５】

ライトバック［５２１］及びリードバック［５２２］を追加する例については、［図４］を参照されたい。

【００２６】

【発明の実施の形態】

本発明は、コンピュータ・プログラム言語用のコンパイラへの追加であり、その基本的な制御流れを［図２］に示す。

【００２７】

ソース・プログラム［３０１］は構文解析系［３１０］により内部表現に変換され、最適化が使用可能であれば、内部表現は最適化プログラム［３２０］により最適化される。最後に、内部形式はバックエンド［３３０］により最終的なオブジェクト・コード［３０２］に変換される。ＳＳＡ形式を使用するコンパイラにおいて、最適化プログラムは少なくとも３つの段階を通常有する。すなわち、’ＳＳＡ前の’内部表現からＳＳＡ形式を使用する内部表現へのプログラムの変換［図１］、ＳＳＡ形式でのプログラムの最適化［３２２］及び、ＳＳＡ形式か

らＳＳＡ形式を使用しない内部表現へのプログラムの変換［３２３］である。通常ＳＳＡ形式は、追加的な操作の存在と表現上のある程度の制約条件のみが非ＳＳＡ内部表現とは異なる。詳細は［ＳＳＡＦＯＲＭ］参照。

【００２８】

使用されるプログラムの望ましい内部表現は次の通りである［図３］。

【００２９】

プログラム［４１０］は一組の関数である。

【００３０】

関数［４２０］は、普通のコンパイラ概念の’基本ブロック’にほぼ対応する一組の’ブロック’［４３０］である。フローグラフは、頂点がブロック［４３０］であり、辺がブロック［４３０］間の制御流れの起こり得る転送であるグラフである。単一のブロック［４３０］は、’エントリブロック’［４２１］として識別され、関数が呼び出されたとき最初に実行される関数内のブロックである。

【００３１】

ブロック［４３０］の中に一連の’命令’［４４０］があり、それぞれの命令は単純な操作を記述している。ブロック［４３０］の中において、制御流れはブロック内の命令の順番と同じ順序で命令［４４０］の間を移動する。制御流れの条件付きの変更は、ブロックに対する後継ブロック［４３２］を選択するときに、どの辺をたどるべきかを選択することによってのみ起きることがある。したがって、ブロックの中の最初の命令［４４０］が実行されれば、他の命令も同様にブロック［４３０］内の順序と同じ順番で実行される。

【００３２】

命令［４４０］は関数呼出しであってもよい、その場合任意の副作用の恐れがあるが、制御流れは関数呼出しに続いて命令［４４０］に最終的に戻らなければならない。

【００３３】

命令［４４０］は’変数’［４５０］を明示的に読み取りあるいは書き込むことができる。各変数は、ソース・プログラム内の局所変数あるいは広域変数のよ

うな’単純変数’［４５１］（あるいはコンパイラにより作られた一時的数値変数）、あるいは他の変数を介して間接的に参照される記憶場所を表す’コンプレックス変数’［４５２］のいずれかである。各変数は、変数の中にどの値を蓄積することができるかを定める型を有する。

【００３４】

コンプレックス変数［４５２］は、’＊（ベース+オフセット）’の形式であり、ここでベース［４５３］は変数［４５０］であり、オフセット［４５４］は一定のオフセットである。この記号式は記憶場所（ベース+オフセット）に蓄積された値を表す。

【００３５】

コンプレックス変数［４５２］を使用したために、計算された記憶場所から値を蓄積あるいは検索するために働く命令［４４０］は通常存在しない。代わりに、ソースあるいはデスティネーションまたはその両方がコンプレックス変数［４５２］である単純なコピーが使用される。同様に、すべての他の命令［４４０］は、コンプレックス変数を使用して、その結果とメモリからのオペランドを蓄積あるいは検索することができる。

【００３６】

プログラム最適化を支援するために、各関数はＳＳＡ形式に変換される。ＳＳＡ形式は「従来の技術」の欄で説明されており、本発明のために修正された場合については、「課題を解決するための手段」の欄で説明されている。この変換はＳＳＡ変換と呼ばれ、３つの段階（ａ）、（ａ’）、（ｂ）で行われる［図１］。

【００３７】

（ａ）［１１０］、［ＳＳＡＦＯＲＭ］で説明したように、ファイ関数が、同じ変数の複数の定義をマージすることができる関数内の任意の位置に挿入される。ファイ関数は、挿入される点において変数の新しい定義を作る。たとえば、ファイ関数［９１０］は、［９１１］（入力プログラムの［８２０］）、［９１２］（入力プログラムの［８３０］）、さらに［１０１０］において、コンプレックス変数’＊ｐ’に書き込まれた異なる値をマージするために挿入され、［１０

１１］（入力プログラムの［８２０］）及び入力プログラムの［８３０］において定められた値をマージする。

【００３８】

この段階のために、プログラム内の任意の点には原変数の実在の定義は１つのみが存在する。

【００３９】

（ａ’）Ｉ．［１２１］、各操作に対して、どの’アクティブな’コンプレックス変数［４５２］が未知の副作用を有する可能性があるかを判定し、リストは操作ノートにこの情報を添付する。これらのノートは以下’可変同期’と呼ばれる。実施例のプログラムにおいて、命令［１０２０］、［１０２１］、［１０２２］及び［１０２３］は、’＊ｐ’をあるいは読み取りまたは修正することがある（それらの命令について何ら情報を有しない）。

【００４０】

ＩＩ．［１２２］、同時に、すべてのコンプレックス変数［４５２］を、元の非ＳＳＡ変数（コンプレックス変数［４５２］に対して記憶場所である）であるそのコンプレックス変数の’同期位置’に、書き戻す必要なライトバックコピー操作［５２１］を追加し、コピー操作自体のデスティネーションにマークを付ける。（これは、ＳＳＡ変換の段階（ｂ）が、新しいＳＳＡ定義としてコピーのデスティネーションを処理することを防止する）。すべてのこのような’ライトバック’［５２１］は、対応する変数を非活動状態にし、したがって、変数が再び定義されない限り、さらにライトバック［５２１］されることをを防止する。

【００４１】

ＩＩＩ．［１２３］、（同期位置に書き戻された後に）無効にされたコンプレックス変数［４５２］の新しいＳＳＡ定義を与えるために、必要なリードバックを追加する。

【００４２】

これはデータフロー問題を解くことにより本質的に行われる。データフロー問題においては、値は’アクティブ・リードバック’である。

【００４３】

アクティブ・リードバックは、上述の段階Ｉで決定したように、コンプレックス変数［４５２］を修正することもある操作によって、あるいは、制御流れのマージ・ポイントにおける同一変数［４５０］の複数のアクティブ・リードバックのマージングによって定められる。この例において、すべての関数呼出しは’＊ｐ’を修正することもあり、したがって［１０２０］、［１０２１］、［１０２２］及び［１０２４］においてリードバックによって表されければならない。

【００４４】

アクティブ・リードバックは、アクティブ・リードバックを有するコンプレックス変数の値を使用する操作によって、あるいはその変数の他のすべてのリードバックがアクティブではない制御流れのマージ・ポイントに到達することによって参照される（そのような拡張定義は、ファイ関数を使用してコンプレックス変数の他の任意の値と次にマージされなければならないから）。

【００４５】

参照されたリードバックのみが、実際にインスタンス生成されなければならない。実施例のプログラムにおいて、唯一のインスタンス生成されたリードバックは［１０３０］にある。インスタンシエーションを生ずる参照は、ソース・プログラム中の［８４０］における変数’ｘ’への’＊ｐ’の割当てである。ＳＳＡ変換されたプログラムにおいて、この割当ては［１０３０］におけるリードバックと［１０３１］におけるファイ関数の間で分割される。

【００４６】

対応するコンプレックス変数［４５２］の定義によって、あるいは変数の新しいリードバックによって強制終了される。この例において、次の関数呼出しが［１０２２］において同一変数の新しいリードバックを定義するから、［１０２１］において定義されたリードバックは強制終了される。

【００４７】

制御流れのマージ・ポイントにおいて、同一変数［４５０］の他のアクティブ・リードバックとマージされ、同一変数の新しいアクティブ・リードバックを生ずる。この例において、’マージ・リードバック’は［１０３０］において定義され、［１０２２］及び［１０２３］における’＊ｐ’のリードバックをマージ

する。

【００４８】

リードバック定義の固定点に到達した後、リードバック変数［４５０］の同期位置から新しいＳＳＡ変数へ値をコピーするために、参照されたものはそれらが定義された場所で適切なコピー操作を挿入することによりインスタンス生成される。必要ならば、この新しい定義点を反映するために新しいファイ関数が挿入されてもよい。上述のように、この例において、これは［１０３０］においてのみ起きる。

【００４９】

段階（a’．Ｉ）［１２１］及び（a’．ＩＩ）［１２２］は次のように行われる。

【００５０】

関数のエントリ・ブロック［４３０］の手順’add_syncs_and_write_backs’［図５］を呼び出し、リストを空にするために、ACTIVE_VARIABLES 及び ALL_ACTIVE_VARIABLES パラメータを初期化する。

【００５１】

引数 BLOCK，ACTIVE_VARIABLES 及び ALL_ACTIVE_VARIABLES を有する手順’add_syncs_and_write_backs’は、次のように定義される［図５］。

【００５２】

［６１０］ブロックのすべての命令［４４０］を実行する。

【００５３】

［６２０］ ALL_ACTIVE_VARIABLES 内の各変数を実行する。

【００５４】

［６２１］命令が変数の読み取りあるいは書き込みであれば、次に［６２２］’可変同期’を加え、予想される参照あるいは修正を命令に記述する。

【００５５】

［６２５］命令が変数の読み取りあるいは書き込みであり、さらに ACTIVE_VARIABLES 内にあれば、変数をその同期位置に書き戻すために、次に［６２６］命令の直前に’ライトバック’コピー操作を加え、［６２７］ ACTIVE_VARIABLES

から変数を除去する。ＳＳＡ変換のこの段階において、（ＳＳＡ変数ではなく）原変数のみが存在するから、このライトバック・コピー操作は変数から変数自身へのコピー（'VARIABLE := VARIABLE'）により表され、デスティネーションはＳＳＡ変換されるべきではないことを示すために特別なフラグが設定される。

【００５６】

［６３０］命令に定義された各変数を実行する。

【００５７】

ACTIVE_VARIABLES 及び ALL_ACTIVE_VARIABLES に変数を加える（これらの変数の修正はこの関数に対して局部的である）。

【００５８】

［６５０］ＤＯＭブロックにより直接支配される各ブロック［４３０］を実行する。

【００５９】

［６５１］支配されているＤＯＭブロック上の add_syncs_and_write_backs を再帰的に使用し、 ACTIVE_VARIABLES 及び ALL_ACTIVE_VARIABLES の局部的な値はそれぞれ名前を付けられたパラメタとして渡される。

【００６０】

段階（a'．ＩＩＩ）は次のように行われる［図６］。

【００６１】

［７０１］マッピング BLOCK_BEGIN_READ_BACKS 及び BLOCK_END_READ_BACKS を空であるように初期化する。これらのマッピングは、フローグラフ内の各ブロックをリードバックのセットに関連づける。

【００６２】

［７０２］待ち行列 PENDING_BLOCKS を関数のエントリ・ブロックに初期化する。

【００６３】

［７１０］ PENDING_BLOCKS が空ではない時に、［７１１］は第一のブロック［４３０］をそれから除去し、そのブロック上の関数' propagate_block_read_backs' ［８００］を呼び出す。

【００６４】

［７２０］’使用中’とマークされたすべてのブロック［４３０］内の各リードバックＲＢに対して、また［７２１］そのソース（それがマージするリードバック）がすべて’使用中’とマークされた’マージ・リードバック’ではないすべてのブロック［４３０］内の各リードバックＲＢに対して、次のようにそのリードバックをインスタンス生成する。

【００６５】

［７３０］ＲＢが’マージ・リードバック’であれば、リードバックの点は［７４１］マージが起きるブロック［４３０］の初めであり、他の場合には［７４２］リードバックを創出した命令［４４０］の直後である。

【００６６】

［７３１］ＲＢの変数をその同期位置からＳＳＡ変数までコピーするリードバックの点でコピー操作を加える（上述のように、ライトバック・コピー操作の追加に対して、ＳＳＡ変数はこの段階では実際には導入されていないから、このコピー操作はその変数からそれ自身へ単純にコピーするが、ＳＳＡ変換をしないようにコピーのソースにフラグをマークする）。

【００６７】

［７３２］必要であれば、新たに定義されたＳＳＡ変数を変数の他の定義とマージするために、ファイ関数を導入する。

【００６８】

パラメータブロックを有する関数’propagate_block_read_backs’は、次のように定義される［図７］。

【００６９】

［８０１］ BLOCK_BEGIN_READ_BACKS 及び、BLOCK_END_READ_BACKS 内のブロックを検索し、対応するリードバック・セットに局所変数 OLD_BEGIN_READ_BACKS 及び OLD_END_READ_BACKS をそれぞれ割り当てる。いずれの場合にもブロックに対する項目がなければ、ブロックに適切な空の項目を追加する。

【００７０】

［８１０］フローグラフ内のブロックの各先行ブロック［４３１］に対する終

わりのリードバックセットの共通部分を計算し、その結果である NEW_BEGIN_READ_BACKS を呼び出す。共通部分は次のように計算される。

【００７１】

他の先行ブロックの１つの中に同一変数のリードバックが存在しないすべての先行リードバックは、結果から廃棄され、さらに’参照済み’とマークされる。

【００７２】

所定の変数に対するリードバックが、すべての先行ブロック［４３１］内の同じリードバックであれば、そのリードバックは結果に追加される。

【００７３】

所定の変数が少なくとも２つの先行ブロック［４３１］内の異なるリードバックにより表されれば、すべての対応する先行リードバックを参照する’マージ・リードバック’が創出され、このマージ・リードバックが結果に追加される。

【００７４】

［８２０］ NEW_BEGIN_READ_BACKS が OLD_BEGIN_READ_BACKS と異なっていれば、あるいはこのブロックが処理されるのが初めてであれば、

［８２１］ NEW_BEGIN_READ_BACKSをBLOCK_BEGIN_READ_BACKS 内のブロックの項目として追加し、 OLD_BEGIN_READ_BACKS を置換する。

【００７５】

［８２２］ NEW_BEGIN_READ_BACKS から NEW_END_READ_BACKS を初期化する。

【００７６】

［８３０］ブロック内の各操作命令に対して、実行する。

【００７７】

［８４０］命令内の各変数レファレンス VREF に対して、実行する。

【００７８】

［８４５］ VREF が NEW_END_READ_BACKS 内に項目ＲＢを有するなら、［８４６］ＲＢを使用中とマークし、［８４７］それを NEW_END_READ_BACKS から除去する。

【００７９】

［８５０］命令内の各変数定義 VDEF に対して、実行する。

【００８０】

［８５５］ VDEF が NEW_END_READ_BACKS 内に項目ＲＢを有するなら、［８５６］ＲＢを NEW_END_READ_BACKS から除去する。

【００８１】

［８６０］可変の変数をあるいは書き込み済みであると指摘する命令内の各可変同期に対して、実行する。

【００８２】

［８６５］ NEW_END_READ_BACKS への変数に対して新しいリードバック・エントリーを追加し、変数のすべての既存のリードバックを置換する。

【００８３】

［８７０］ NEW_END_READ_BACKS が、OLD_END_READ_BACKS と異なれば、

［８７１］ NEW_END_READ_BACKS を BLOCK_END_READ_BACKS 内のブロックの項目として追加し、 OLD_END_READ_BACKS を置換する。

【００８４】

［８８０］各ブロックの後継［４３２］を PENDING_BLOCKS に追加する。

【００８５】

（ｂ）［１３０］［ＳＳＡＦＯＲＭ］で前述のように、すべての非ＳＳＡ変数定義がユニークなＳＳＡ変数の定義により置換され、すべての非ＳＳＡ変数レファレンスが適切なＳＳＡ変数へのレファレンスにより置換される。

【００８６】

この規則の例外は、段階（ａ’）で挿入されたコピー命令［４４０］内で特別な’同期’位置としてマークされたコンプレックス変数［４５２］である。それらはそのままにしておかれ、元のコンプレックス変数［４５２］が参照される。

【００８７】

本発明を使用したものと、本発明を使用しないものの、ＳＳＡ形式に変換されるプログラムの例を図８から図１１に示す。

【００８８】

【発明の効果】

本発明は、別名が付けられている可能性のある状態で、非局所的メモリ位置に

対して、ＳＳＡ形式の効率的な使用を可能にする同期操作を追加する。

【図面の簡単な説明】

【図１】

本発明に使用されるＳＳＡ変換処理の一般的な形式を示す。

【図２】

コンパイラの総合的な制御流れを示す。

【図３】

本発明の説明に使用された基本的なデータ構造を示す。

【図４】

変数リードバック及びライトバックの配置を示す。

【図５】

修正されたＳＳＡ変換処理の段階（ａ’．Ｉ）及び（ａ’．ＩＩ）に対する手順の制御流れを示し、命令に変数の同期情報を追加し、変数ライトバックを関数’add_syncs_and_write_backs’に追加する。

【図６】

修正されたＳＡ変換処理の段階（ａ’．ＩＩＩ）に対する手順の制御流れを示し、関数に変数リードバックを追加する。

【図７】

修正されたＳＳＡ変換処理’add_merged_read_backs’の段階（ａ’．ＩＩＩ）により使用されたサブルーチンに対する制御流れを示す。

【図８】

ソース・プログラムの例を示す。

【図９】

リードバックの単純な具体化例を有するＳＳＡ変換されたプログラムを示す。

【図１０】

本発明において説明されたリードバックの具体化例を有するＳＳＡ変換されたプログラムを示す。

【図１１】

レジスタ割り当てされＳＳＡ変換されていないプログラムを示す。

【図１２】

従来のＳＳＡ変換処理の一般的な形式を示す。

【符号の説明】

３０１　入力ソースファイル

３０２　出力ファイル

３２０　最適化プログラム

４１０　プログラム

４２０　関数

４２１　エントリブロック

４３０　ブロック

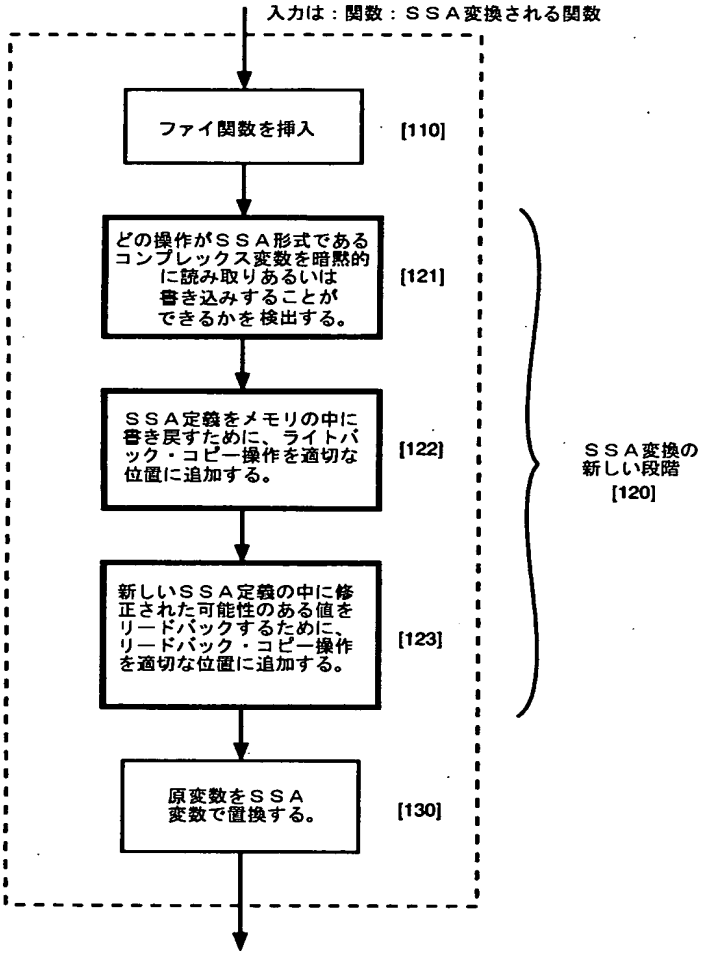
４４０　命令

４５０　変数

【書類名】　　図面

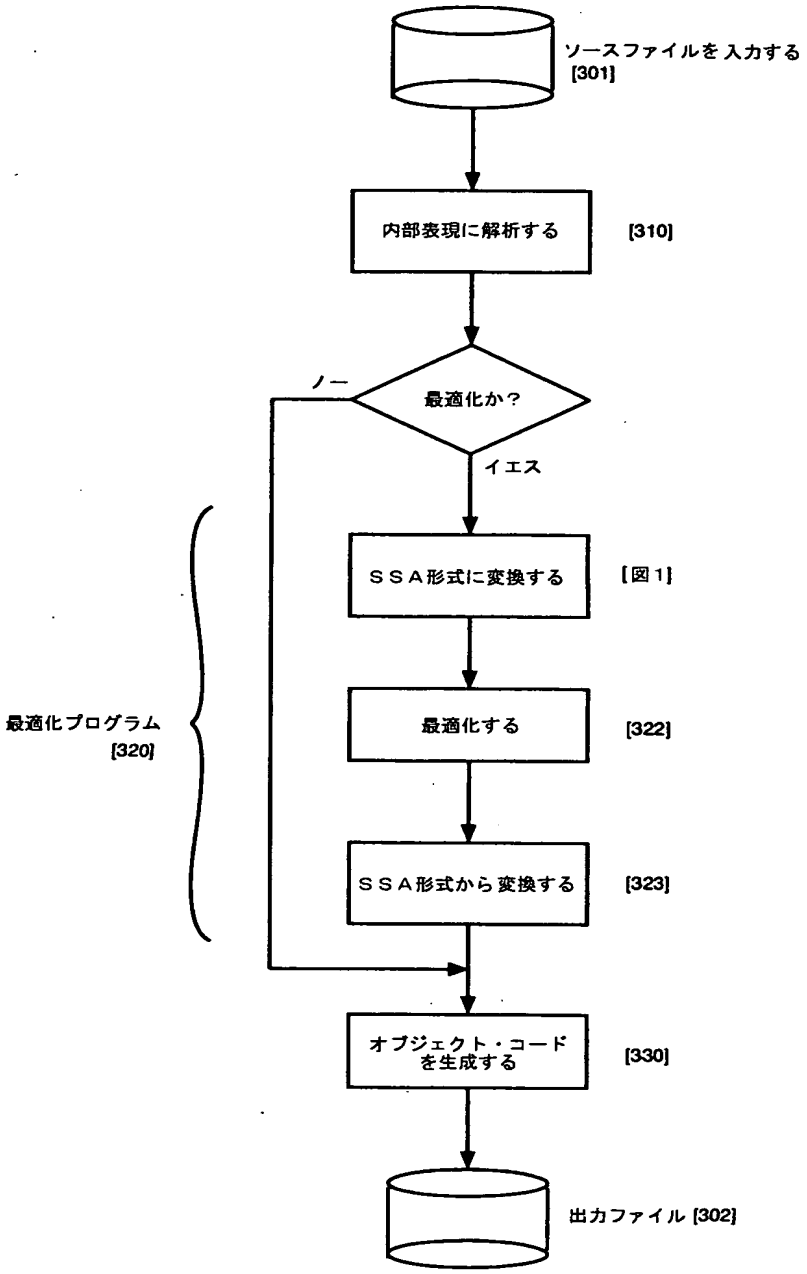
【図１】

修正されたSSA変換処理



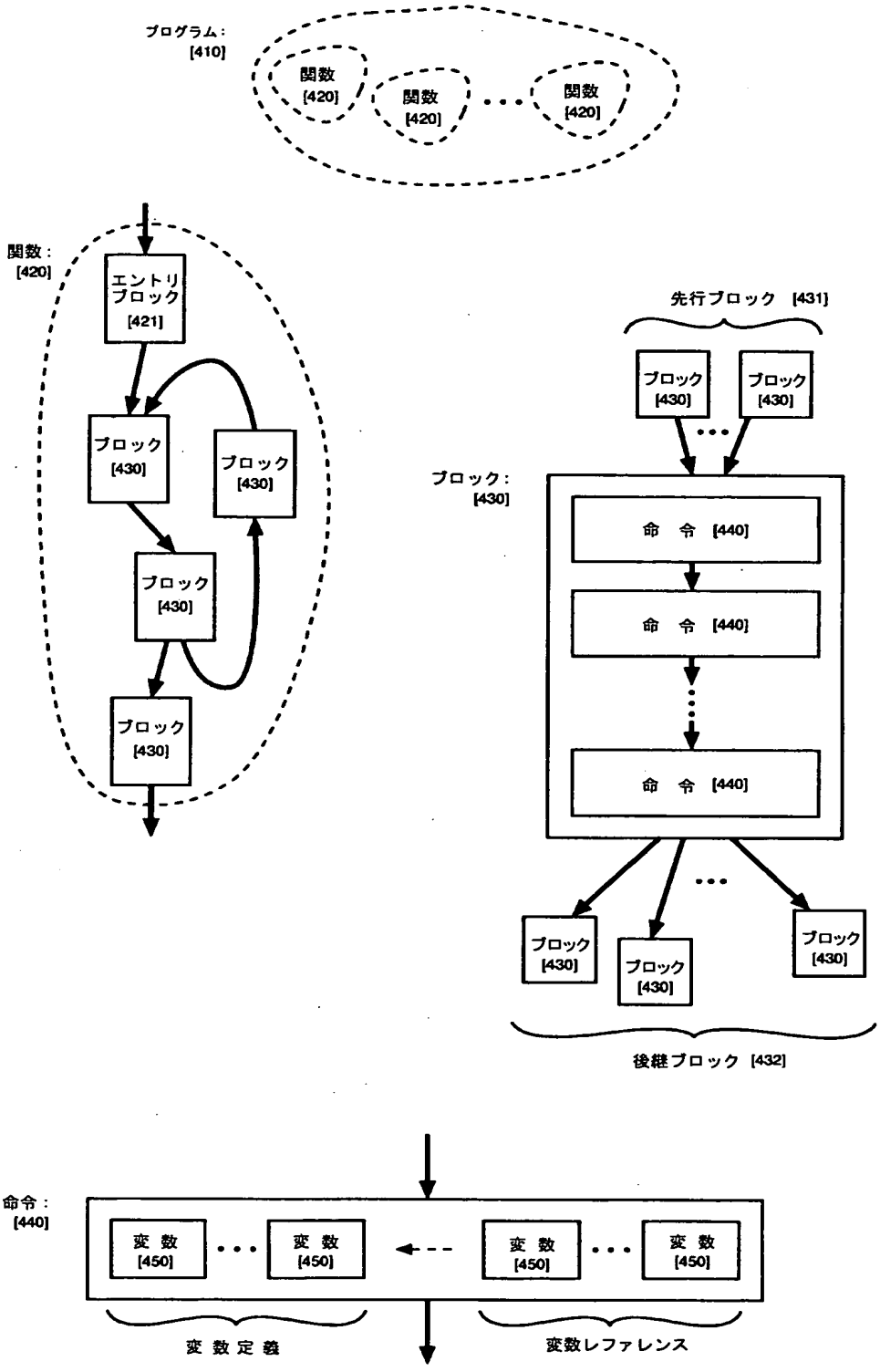
【図２】

総合的なコンパイラ 制御流れ



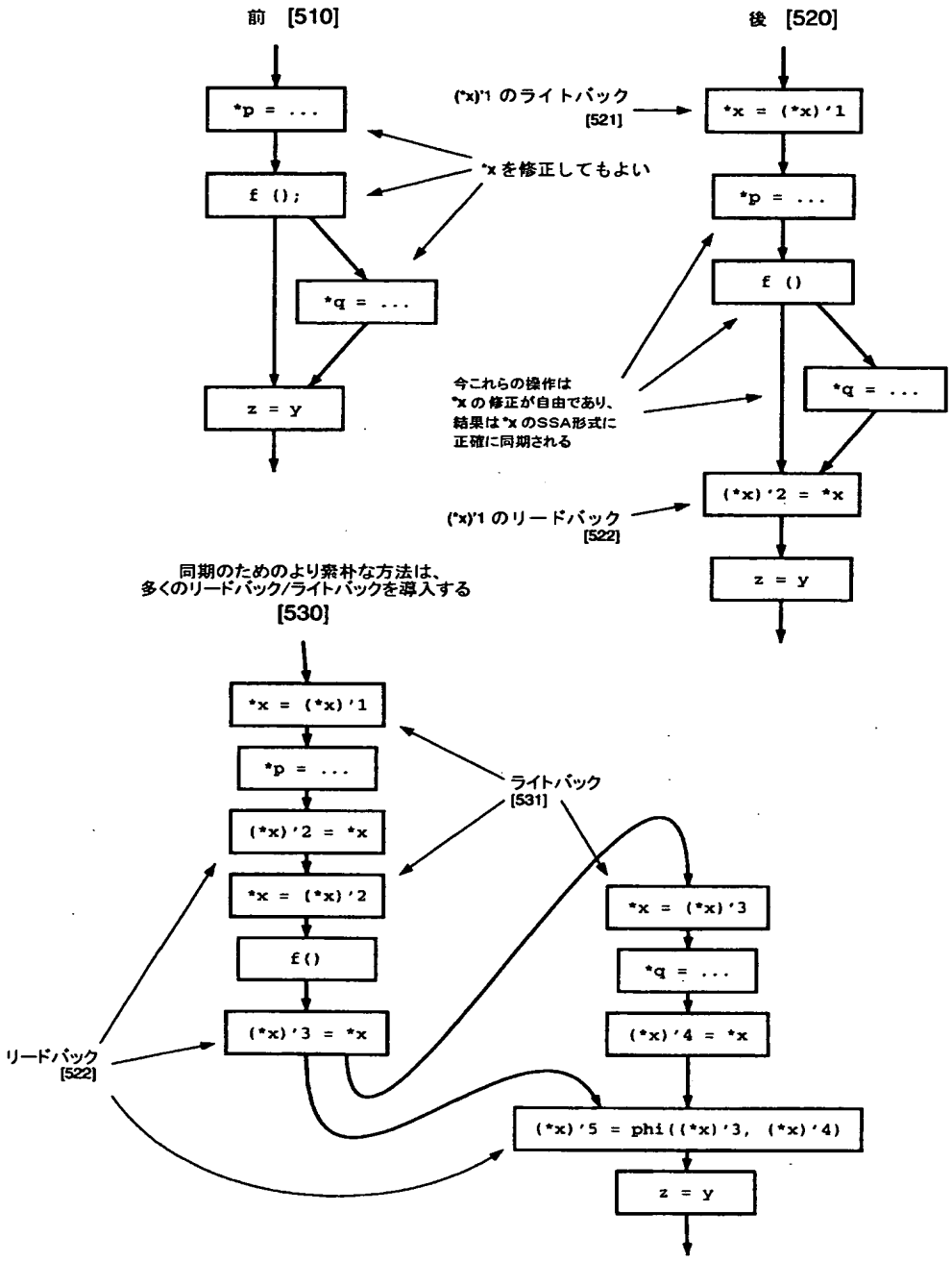
【図３】

プログラム表現

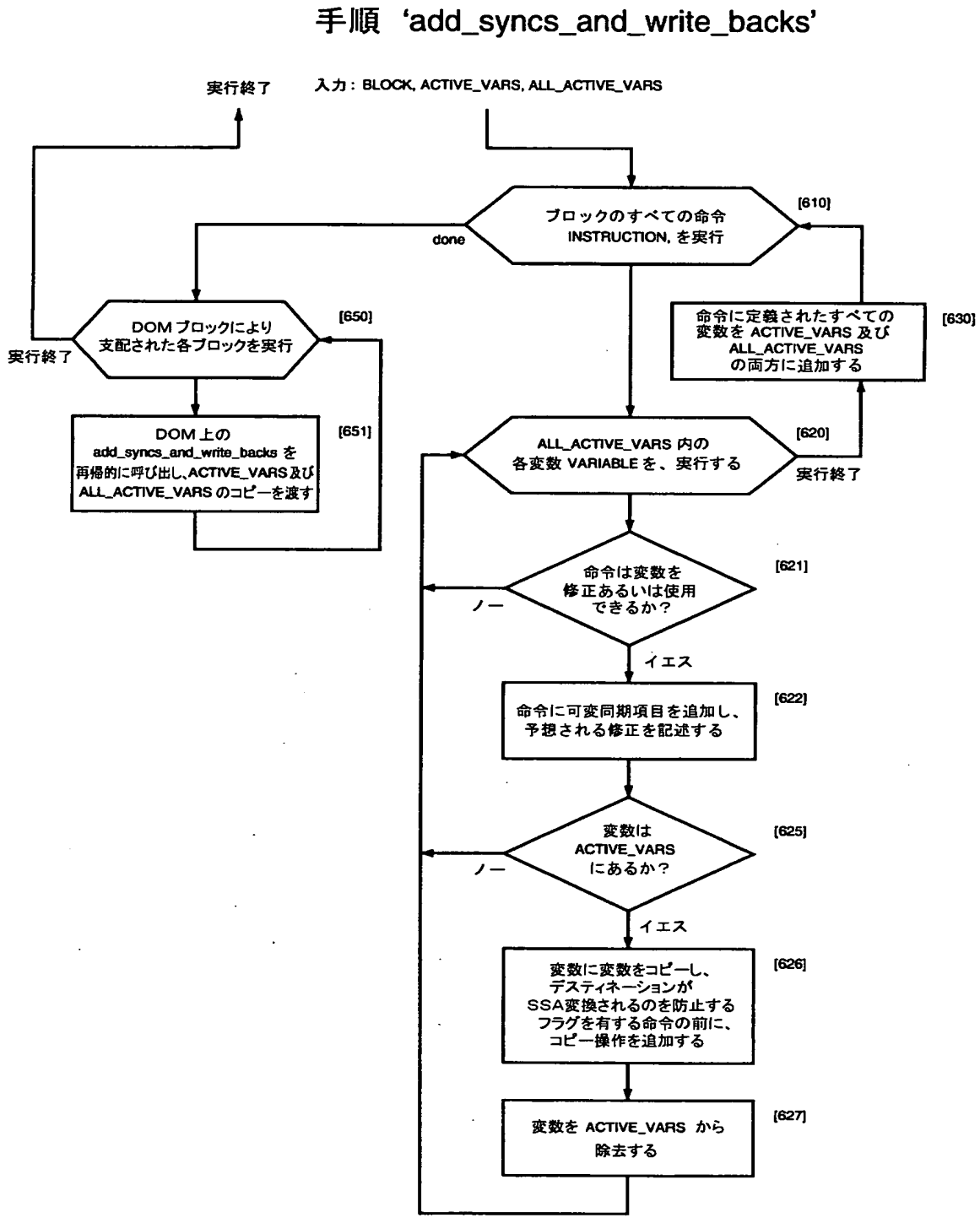

プログラム：
[410]
関数
[420]
関数
[420]
関数
[420]
関数：
[420]
エントリ
ブロック
[421]
ブロック
[430]
ブロック
[430]
ブロック
[430]
ブロック
[430]
先行ブロック [431]
ブロック
[430]
ブロック
[430]
ブロック：
[430]
命 令 [440]
命 令 [440]
命 令 [440]
ブロック
[430]
ブロック
[430]
ブロック
[430]
後継ブロック [432]
命令：
[440]
変 数
[450]
変 数
[450]
変 数
[450]
変 数
[450]
変 数 定 義
変数レファレンス

【図４】

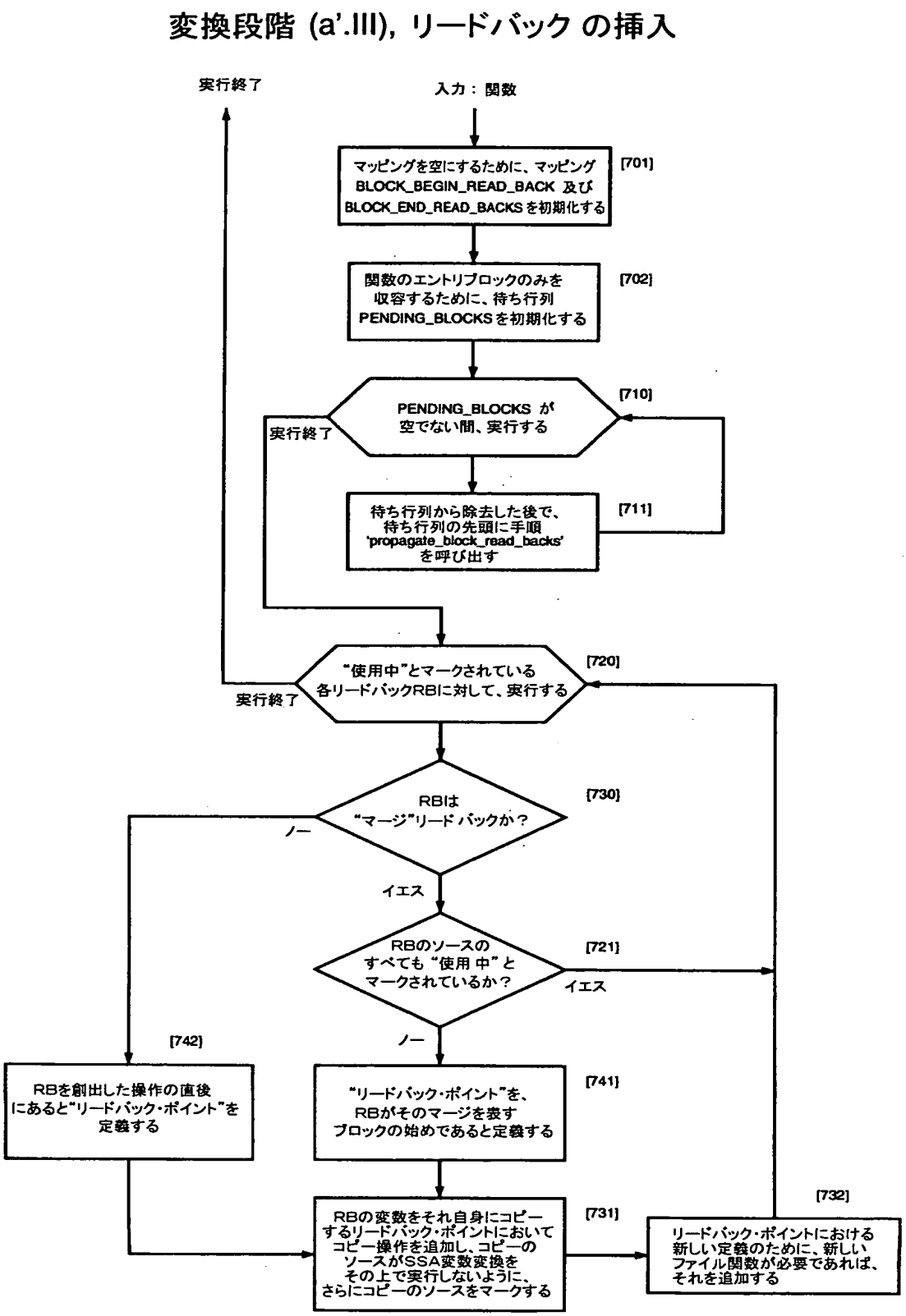
*x, (*x)'1 のＳＳＡ形式に対するリードバック/ライトバックの配置



【図５】

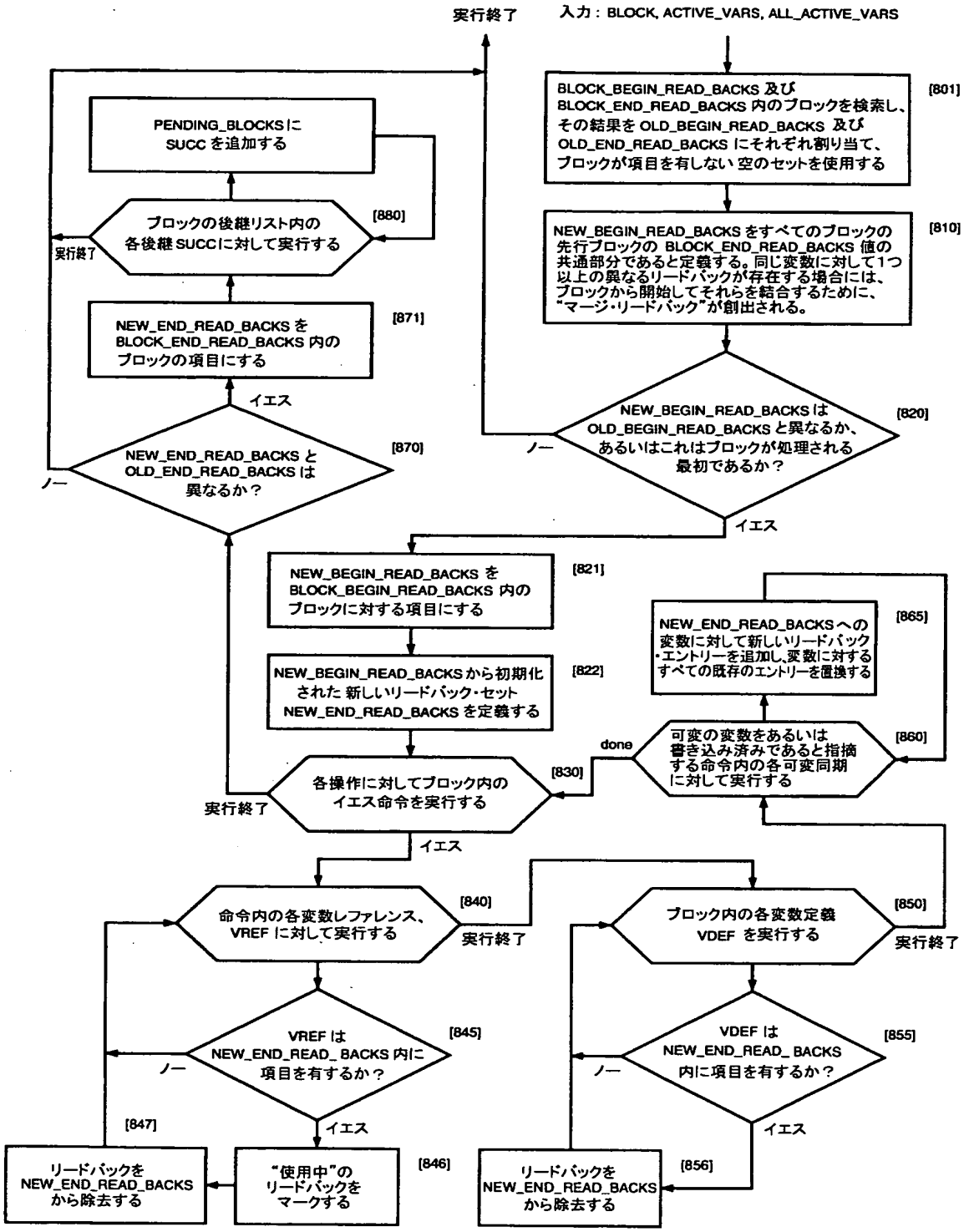

手順 'add_syncs_and_write_backs'
実行終了
入力：BLOCK, ACTIVE_VARS, ALL_ACTIVE_VARS
ブロックのすべての命令 INSTRUCTION, を実行
[610]
done
DOM ブロックにより 支配された各ブロックを実行
[650]
実行終了
DOM 上の add_syncs_and_write_backs を 再帰的に呼び出し、ACTIVE_VARS 及び ALL_ACTIVE_VARS のコピーを渡す
[651]
命令に定義されたすべての 変数を ACTIVE_VARS 及び ALL_ACTIVE_VARS の両方に追加する
[630]
ALL_ACTIVE_VARS 内の 各変数 VARIABLE を、実行する
[620]
実行終了
命令は変数を 修正あるいは使用 できるか？
[621]
ノー
イエス
命令に可変同期項目を追加し、 予想される修正を記述する
[622]
変数は ACTIVE_VARS にあるか？
[625]
ノー
イエス
変数に変数をコピーし、 デスティネーションが SSA変換されるのを防止する フラグを有する命令の前に、 コピー操作を追加する
[626]
変数を ACTIVE_VARS から 除去する
[627]

【図６】


変換段階 (a'.III), リードバック の挿入
実行終了
入力：関数
マッピングを空にするために、マッピング BLOCK_BEGIN_READ_BACK 及び BLOCK_END_READ_BACKS を初期化する
[701]
関数のエントリブロックのみを収容するために、待ち行列 PENDING_BLOCKS を初期化する
[702]
PENDING_BLOCKS が空でない間、実行する
[710]
実行終了
待ち行列から除去した後で、待ち行列の先頭に手順 'propagate_block_read_backs' を呼び出す
[711]
"使用中"とマークされている各リードバックRBに対して、実行する
[720]
実行終了
RBは"マージ"リードバックか？
[730]
ノー
イエス
RBのソースのすべても"使用中"とマークされているか？
[721]
イエス
ノー
[742]
RBを創出した操作の直後にあると"リードバック・ポイント"を定義する
"リードバック・ポイント"を、RBがそのマージを表すブロックの始めであると定義する
[741]
RBの変数をそれ自身にコピーするリードバック・ポイントにおいてコピー操作を追加し、コピーのソースがSSA変数変換をその上で実行しないように、さらにコピーのソースをマークする
[731]
リードバック・ポイントにおける新しい定義のために、新しいファイル関数が必要であれば、それを追加する
[732]

【図７】

手順 'propagate_read_backs'



【図８】

ソース・プログラムの例

この短いCプログラムは本発明を説明するために 使用される ：

```
extern int g (), h (), i (), x;
int foo (int *p)
{
  (*p)++;                                          [810]
  if (*p > 10)
    {
      g ();
      h ();
      if (x > 5)
        g ();
      if (x > 3)
        i ();
      else
        x = *p;
      *p = 5;
    }
  return *p;
}
```

やや原始的な形式に変換された同じプログラムを 次に示す ：

```
int foo (int *p)
{
block1:
  *p := *p + 1;                                    [820]
  if (*p <= 10)
    goto block8;

block2:
  g ();
  h ();
  if (x <= 5)
    goto block4;

block3:
  g ();

block4:
  if (x > 3)
    goto block6;

block5:
  x := *p;                                         [840]
  goto block7;

block6:
  i ();

block7:
  *p := 5;                                         [830]

block8:
  return *p;
}
```

【図９】

リードバックの単純な例を有するＳＳＡ変換されたプログラム

以下は、「ファイ」操作で強化した擬似C であり、ここで、

```
RESULT = phi (block1: VAL1, ..., blockN: VALN)
```

は、「制御流れがブロック１から来るならば、VAL1 を RESULT に割り当てる」
ことを意味し、Nの各値に対して以下同様である

Nが整数である追加の変数「pvN」は、*p のSSAバージョンであり、
実際には局所変数であって、pのデレファレンスではない

```
int foo (int *p)
{
  int pv1, pv2, pv3, pv4, pv5, pv6;

block1:
  pv1 = *p + 1;
  if (pv1 <= 10)
    goto block8;

block2:
  *p = pv1;          /* これは PV1 を *P へライトバックする  */
  g ();
  pv2 = *p;          /* これは *P を PV2 へリードバックする   */
  *p = pv2;          /* これは PV2 を *P へライトバックする  */
  h ();
  pv3 = *p;          /* これは *P を PV3 へリードバックする   */          [912]
  if (x <= 5)
    goto block4;

block3:
  *p = pv3;          /* これは PV4 を *P へライトバックする  */
  g ();
  pv4 = *p;          /* これは *P を PV4 へリードバックする   */          [911]

block4:
  pv5 = phi (block3: pv4, block2: pv3)                                   [910]
  if (x > 3)
    goto block6;

block5:
  goto block7;

block6:
  i ();

block7:
  x = phi (block6: x, block5: pv5);

block8:
  pv6 = phi (block1: pv1, block7: 5);
  *p = pv6;          /* これは PV6 を *P へライトバックする  */

  return pv6;
}
```

【図１０】

本発明で説明されたリードバックの例を有するＳＳＡ変換されたプログラム

```
int foo (int *p)
{
  int pv1, pv2, pv3;

block1:
  pv1 = *p + 1;                                                    [1011]
  if (pv1 <= 10)
    goto block8;

block2:
  *p = pv1;          /* これはPV1を*Pへライトバックする  */
  g ();                                                            [1021]
  h ();                                                            [1022]
  if (x <= 5)
    goto block4;

block3:
  g ();                                                            [1023]

block4:
  pv2 = *p;          /* これは*PをPV2へリードバックする  */        [1030]
  if (x > 3)
    goto block6;

block5:
  goto block7;

block6:
  i ();                                                            [1024]

block7:
  x = phi (block6: x, block5: pv2);                                [1031]

block8:
  pv3 = phi (block1: pv1, block7: 5);                              [1010]
  *p = pv3;          /*  これはPV3を*Pへライトバックする   */

  return pv3;
}
```

【図１１】

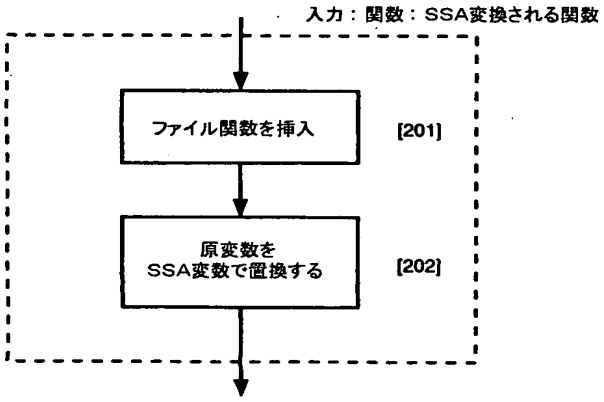
レジスタ割付けされ、ＳＳＡ変換されていないプログラム

良いレジスタ・アロケータは短い生存期間を有する多くの変数を生成する傾向があるから、ＳＳＡ形式を使用するためには、可能な限り変数をマージする良いレジスタ・アロケータを有する必要がある

```
int foo (int *p)
{
  int pv;

block1:
  pv = *p + 1;
  if (pv <= 10)
    goto block8;

block2:
  *p = pv;            /*  これはPVを*Pへライトバックする  */
  g ();
  h ();
  if (x <= 5)
    goto block4;

block3:
  g ();

block4:
  if (x > 3)
    goto block6;

block5:
  x = *p;
  goto block7;

block6:
  i ();

block7:
  pv = 5;

block8:
  *p = pv;            /*  これはPVを*Pへライトバックする  */

  return pv;
}
```

【図１２】

従来のSSA変換処理



【書類名】　　要約書

【要約】

【課題】　’ＳＳＡ形式’として知られるコンパイラ表現の通常の公式化は単に局所変数のみを処理できる。他の位置の表現を可能とするために、これを拡張することが望ましい。

【解決手段】　本発明は、別名が付けられている可能性のある状態で、非局所的メモリ位置に対して、ＳＳＡ形式の効率的な使用を可能にする同期操作を追加する。

【選択図】　　図１

出 願 人 履 歴 情 報

識別番号　〔０００００４２３７〕

１．変更年月日　１９９０年　８月２９日
［変更理由］　新規登録
住　所　東京都港区芝五丁目７番１号
氏　名　日本電気株式会社